UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS
Departamento de Sistemas Informáticos y Computación.

FORMALIZANDO EL PROCESO DE DEPURACIÓN EN PROGRAMACIÓN FUNCIONAL PARALELA Y PEREZOSA.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Alberto de la Encina Vara

Bajo la dirección de los doctores
Luís Fernando Llana Díaz
Fernando Rubio Díez

Madrid, 2008

- **ISBN: 978-84-692-1748-1**

Formalizando el proceso de depuración en programación funcional paralela y perezosa

Tesis Doctoral

Alberto de la Encina Vara

Departamento de Sistemas Informáticos y Computación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Formalizando el proceso de depuración en programación funcional paralela y perezosa

Memoria presentada para obtener el grado de
Doctor en Ciencias Matemáticas
Alberto de la Encina Vara

Dirigida por los profesores
Luis Fernando Llana Díaz
Fernando Rubio Diez

Departamento de Sistemas Informáticos y Computación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Índice general

Agradecimientos

Considero la época dedicada a la tesis como un cuento. En ocasiones dicho cuento parecía un cuento de terror, en otras un cuento de desengaños y amores, a veces parecía el cuento sin fin, pero por fin dicho cuento ha terminado. La verdad es que para el autor es un alivio que esto ya haya sucedido, pues la lucha por alcanzar este final ha sido tortuosa y llena de obstáculos. A lo largo de este cuento han ido apareciendo diversos personajes, algunos que han ayudado más que otros, algunos aparecieron momentáneamente y desaparecieron, otros se mantuvieron perennes durante todo el proceso, como si fueran parte del paisaje o ayudando a que se realizara el entramado del cuento.

En principio, quisiera agradecer a aquellas personas que se mantuvieron perennes durante todo el proceso, como si fueran la voz de un dios que ayudaba por detrás, aunque en ocasiones frustraban un poco con las exigencias, a la realización de esta tesis. Éstos han sido mis directores de tesis: Fernando Rubio Diez y Luis Fernando Llana Díaz.

Algunos se encuentran en los comienzos del cuento. Entre ellos cabe destacar a Ricardo Peña, que fue mi tutor del trabajo de tercer ciclo y en el que gran parte de esta tesis está basado. Es más, el trabajo de tercer ciclo se puede considerar como un preámbulo de esta tesis, ya que gran parte de los métodos y desarrollos científicos son similares.

Uno de los grandes personajes principales de este cuento ha sido mi madre. Siempre se encontraba ahí pendiente de que finalizara esta tesis y según ella con más ganas que yo de que la terminara (cosa que sigo dudando). He de agradecer de forma sincera y con un cariño especial todo ese apoyo.

También he de agradecer a otro personaje importante que apareció en este cuento: mi novia, Marta. Me conoció desarrollando esta tesis y todavía nunca me ha conocido con ella finalizada. Supo esperar con paciencia a que yo terminara el cuento, para comenzar uno nuevo junto a ella. Espero que ahora que la finalizo podamos llevar a cabo más proyectos juntos y "la otra" (como ella llamaba a la tesis) no sea un inconveniente.

También he de agradecer la ayuda de Hans Wolfgang Loidl, que un día pasaba por aquí y sin que él lo pidiera le di un papel de relevancia en dicho cuento. Apareció porque yo lo quise, le pedí ayuda para realizar la implementación de la versión paralela del depurador y me dio interesantes ideas de cómo se podía implementar. Fue de gran utilidad y me ayudó a elegir el camino adecuado.

Dentro de la gran cantidad de personas que han aparecido en este cuento quiero nombrar de forma especial, a mis hermanos, que siempre han estado preocupados por mí, a Mercedes Hidalgo Herrero, con la que tantas veces discutí de detalles técnicos y según ella siempre ganaba el más cabezón (es decir, yo), a Olga Marroquín Alonso, con la que tantas conversaciones he tenido y que siempre ha estado preocupada por el progreso del cuento, a Natalia López Barquilla, que

siempre animaba y mantenía una sonrisa, a Ismael Rodríguez Laguna, que me aguantó como compañero de despacho y además se ofreció a ayudar muchas veces, a Manuel Núñez García, ya que esta tesis está subvencionada por los proyectos conseguidos bajo su dirección y aunque la primera vez que le pedías algo siempre gritaba, sólo era necesario esperar a perdírselo por segunda vez, a Francisco Javier Crespo Yáñez, que tantas veces vino a dar cháchara, y ya por último a las niñas de Luis Llana, Isabel y Almudena, que cada vez que estaba con ellas conseguía que me olvidara del cuento.

Por último quisiera agradecer a todas aquellas personas que entraron en el cuento, me animaron y salieron de él, "que ni qué ni qué" que dejaron una gran huella, sin esperar nada a cambio. Para ellos este cuento fue como un momento en el que entraron, realizaron su papel auxiliar y ssssssuuuuuuuuuuu... se fueron. Para mí, sin embargo, fue como una aventura dentro del cuento en la que encontraba una mano amiga que no esperaba nada a cambio.

La realización de este cuento influyó mucho en mi vida personal, la mayoría de veces negativamente. Pero también he de comentar que yo comencé este cuento viviendo con mi familia y sin novia y lo termino, con casa, con novia y viviendo con ella. Creo que una nueva aventura comienza, pero eso es otra historia, seguro más complicada que ésta, que aquí no contaré y que probablemente nunca escriba.

Lo último que quería hacer en esta sección de agradecimientos es agradecer al autor de esta tesis que por fin la terminara. En ciertas ocasiones consideré que nunca sería capaz, va a ser todo un alivio para mí...

Capítulo 1

Introducción

¿Cuántas veces nos planteamos por qué investigamos en lo que estamos investigando? En concreto, cuántas veces nos habremos y nos habrán preguntado "¿por qué la programación funcional funciona?" y "¿qué mejoras introduce?". Fueron tantas veces, que en 1989 John Hughes publicó el artículo [Hug89] en el que respondía directamente a dicha pregunta. Este artículo fue posteriormente incluido como capítulo de libro [Hug90]. Este artículo ha sido traducido a muchos idiomas incluyendo al japonés y al chino. Esto nos puede dar una primera impresión de la relevancia que tiene la programación funcional. En este artículo se presenta de forma clara y con ejemplos la facilidad con la que se programa en un entorno funcional debido entre otros a la ausencia de efectos laterales, el orden superior, el polimorfismo, su semántica simple, la facilidad para realizar transformaciones de forma segura y la evaluación perezosa.

De la exposición anterior se deduce claramente que esta tesis va a tener como su pilar central la programación funcional. Otros pilares de esta tesis corresponden con la depuración, los métodos formales y el paralelismo. A continuación iremos estructurando la tesis que se presenta en base a dichos pilares, de tal forma que podamos hacernos una idea inicial de la construcción completa.

Como ya se ha comentado, la programación funcional corresponde con nuestro pilar central de la tesis. Aunque la programación funcional posea ciertas ventajas [Bir98] con respecto a la imperativa, la distancia entre esta y la máquina Von Neumann es muy grande. Es por eso que a lo largo de la historia se han ido desarrollando varias máquinas abstractas que acercaron la programación funcional a la arquitectura de los computadores. Entre ellas destacaremos las máquinas de Sestoft [Ses97] y la máquina abstracta *STG* [PS89, Pey92] (Spineless Tagless G-Machine). Destacamos las máquinas de Sestoft ya que presentan la evolución de la pereza de forma clara y sencilla. Aunque no han sido derivadas automáticamente de la semántica, se puede considerar que se encuentran muy cerca de ser derivadas de ella y, por tanto, la comprensión de sus pasos de evaluación es sencilla. No obstante no se las puede considerar como unas máquinas abstractas óptimas y por ello no han sido implementadas. Casi en el otro extremo nos encontramos con la máquina abstracta *STG* que es una máquina abstracta optimizada e implementada en el compilador GHC y, por tanto, esta máquina abstracta sigue viva y mantenida. Recientemente ha cambiado el modelo de evaluación de las funciones pasando del modelo *push/enter* al modelo *eval/apply* [MP04, MP06]. En el trabajo [EP08] desarrollado por el autor de esta tesis se estudian las diferencias entra ambos métodos de evaluación y se derivan unos esquemas de traducción a código semi-imperativo.

Otro de los pilares de esta tesis corresponde con la depuración. En un mundo utópico don-

de los programas se realizan correctamente, esta sería innecesaria y, por tanto, esta tesis no tendría mucho sentido. Desafortunadamente, o afortunadamente para nosotros, los programas suelen poseer errores y es necesario corregirlos. Los depuradores son herramientas que ayudan al programador a la hora de localizar los errores de programación. Al ser la programación funcional más abstracta que la imperativa, la cantidad de errores que se producen es menor, pero a medida que los programas crecen, la posibilidad de provocar un error suele incrementarse. Es por eso que los depuradores también son necesarios en programación funcional. Sin embargo los primeros depuradores en este campo tardaron bastante en surgir. Esto es debido a que la pereza complica mucho el mecanismo de depuración de los programas. En esta tesis consideraremos un modelo de depuración similar al de Hood [Gil00a, Gil00b], pues lo consideramos sencillo de utilizar, ya que está basado en anotar las expresiones de las que queremos observar sus resultados.

El paralelismo es otro de los pilares de esta tesis. Normalmente no suele ser sencillo desarrollar programas que evalúen de forma eficiente en paralelo y, al igual que en otras areas de programación, en funcional también han surgido distintos lenguajes de programación funcional paralela. Nosotros en esta tesis nos centraremos en dos de ellos: GpH [THJ+96, THLP98] y Eden [BLOP96, BLOP98, PR01, LOP05]. Ambos lenguajes de programación funcional paralela, extienden el lenguaje perezoso Haskell, y con poco trabajo de programación dan como resultado aceptables tasas de aceleración.

El último pilar que queda por comentar corresponde con los métodos formales. Estos han sido tantas veces criticados porque sus resultados son difícilmente aplicables a un mundo real, como defendidos, ya que se les puede considerar como la base de los métodos aplicados al mundo real. Nuestra opinion al respecto es simple y consiste en que "hacer las cosas bien nunca está de más". Si bien es cierto que a veces los métodos formales desarrollan teorías que están inicialmente lejos de poderse utilizar en la programación, consideramos estos como base del desarrollo de futuras aplicaciones. Nosotros entenderemos los métodos formales, en esta tesis, como un método de demostrar que el desarrollo llevado a cabo es correcto.

Una vez que ya tenemos los pilares básicos de nuestra tesis, creemos conveniente explicar en detalle las uniones entre ellos. El depurador Hood fue implementado en el año 2000 y como su autor reconoce *es conveniente presentar la semántica de dicho depurador de una forma clara*, ya que debido a la pereza el comportamiento de Hood resulta a veces muy difícil de comprender. Esto nos motivará para estudiar todas las posibles implementaciones que se han realizado de dicho depurador y a darle una semántica al estilo de la semántica de Sestoft.

Nuestra meta no está centrada en esta tarea, sino que pretendemos generar el primer depurador que funcione en dos lenguajes paralelos perezosos (GpH y Eden), para lo cual realizaremos una versión paralela de Hood. Por este motivo nos parece interesante iniciar el camino comprendiendo el comportamiento de dicho depurador. Esta semántica nos servirá de guía para la realización tanto del depurador en paralelo, como de su semántica en paralelo.

Una vez que hayamos implementado dicho depurador en el entorno paralelo nos planteamos pasar a desarrollar un nuevo análisis para el lenguaje funcional Eden; el análisis de *especulación*. Desarrollaremos una herramienta que permita ver gráficamente la especulación que se genera en cualquier programa de Eden a lo largo del tiempo.

Todos estos pasos pretendemos realizarlos desde una perspectiva formal. Demostrando en cada caso que la depuración no interfiere con la evaluación del programa y que cuando se evalúa una clausura anotada con una observación se obtiene una observación.

Resumiendo, las principales aportaciones prácticas de esta tesis corresponderán con la imple-

mentación del primer depurador para dos lenguajes paralelos perezosos y utilizar dicho depurador para un nuevo análisis, el análisis de especulación en el lenguaje Eden. Este análisis es interesante, ya que la especulación puede llevar a que se realicen más tareas de las necesarias para el resultado final del programa y, por tanto, tardar más tiempo en finalizar el programa y enviar más datos de los necesarios entre los procesos.

Una vez explicada la relación entre los pilares básicos y las pretensiones de esta tesis, pasaremos a explicar la estructura general de esta tesis. Los capítulos originales de esta tesis se presentan a partir del Capítulo 5. En los primeros capítulos se realiza un resumen del área necesaria para el desarrollo de la tesis. La estructura de la tesis se detalla a continuación.

En el Capítulo 2 se presenta un resumen tanto de la semántica de la programación funcional perezosa, como de las máquinas abstractas que se han utilizado para su implementación. En concreto se presenta la semántica natural de Launchbury y la posterior modificación de Sestoft, así como un resumen de las máquinas abstractas presentadas por Sestoft, junto con una de sus optimizaciones: la poda de entornos. También se presenta la máquina abstracta *STG*, que será parte imprescindible para la comprensión del Capítulo 5 y, finalmente, se presentan las ideas principales de la extensión que suele realizarse tanto en las semánticas como en las máquinas abstractas referente a los enteros primitivos, que se utilizará a lo largo de esta tesis.

Posteriormente, en el Capítulo 3 se presenta un resumen de las diferentes líneas de depuración que se han generado a lo largo de la historia en programación funcional perezosa. Comentaremos de forma resumida los depuradores más relevantes, destacando sus características más significativas, para después centrarnos en el depurador Hood. De este depurador no sólo presentamos un resumen, sino varios detalles que serán utilizados a lo largo de esta tesis.

En el Capítulo 4 presentamos un resumen de los lenguajes funcionales paralelos más relevantes que se utilizarán para el desarrollo posterior de los capítulos 7 y 8. Entre ellos destacamos y presentamos de forma breve *GpH* y *Eden*, que son los lenguajes paralelos sobre los que vamos a extender la depuración. Nótese que los contenidos del Capítulo 4 no serán utilizados hasta llegar al Capítulo 7. En consecuencia, el lector puede elegir entre leer los capítulos en el orden en el que aparecen en la tesis o no leer el Capítulo 4 hasta justo antes de empezar el Capítulo 7.

A partir de la finalización del Capítulo 4 es cuando comienza el trabajo original de esta tesis. En el Capítulo 5 se presentan las principales ideas de la formalización de un depurador al estilo de Hood a nivel de la máquina *STG* junto con las diferentes opciones implementadas de dicho depurador y con las posibles mejoras que podrían realizarse trabajando a nivel de la máquina *STG*. Existen al menos dos versiones de dicho depurador, y el comportamiento de ambas versiones no es completamente equivalente. Es por ese motivo que su formalización a nivel de dicha máquina abstracta nos clarificará dichas diferencias. Por otro lado, en este capítulo se propone una mejora en el depurador Hood consistente en la compartición de clausuras. En este capítulo nos sentimos libres para desarrollar todas estas versiones de una forma semi-formal, pero no pretendemos demostrar ninguna propiedad con respecto a la implementación. Simplemente hay que considerarlo como una guía que nos permite entender de forma clara el comportamiento de Hood y sus diferentes versiones, así como otras posibles opciones que podrían adoptarse. Además se han implementando varios intérpretes que nos han permitido desarrollar los ejemplos de forma automática.

El siguiente capítulo, Capítulo 6, presenta una semántica de un lenguaje funcional perezoso con marcas de observación como las de Hood. Este capítulo se desarrolla desde una perspectiva rigurosa y formal. Se parte de un λ-cálculo básico extendido con expresiones **letrec**, **case**, cons-

tructores y valores primitivos, al que se le han incorporado anotaciones de observación. Es en este punto donde se crea una semántica para dicho lenguaje y se prueba que dichas marcas de observación no modifican el comportamiento del programa, es decir, la semántica es equivalente a la original sin observaciones, y genera observaciones cuando se entra en una clausura etiquetada como observable. Para su mejor comprensión se presentan varios ejemplos de evaluación y las demostraciones formales se incluyen en el Apéndice A. Una vez demostradas dichas propiedades de la semántica y siguiendo el trabajo de Sestoft, se desarrolla formalmente una máquina abstracta que incluye la depuración. Se demuestra su corrección y equivalencia con respecto a la semántica y se presenta un ejemplo de evaluación en dicha máquina y, por tanto, equivalente a la máquina de Sestoft sin observaciones.

En el Capítulo 7 se presenta la extensión paralela de la librería de depuración Hood a los entornos de programación *GpH* y *Eden*. Este capítulo consiste en una extensión al entorno paralelo de la librería Hood junto con un caso de uso de dicha librería para el análisis de especulación en Eden. Los detalles técnicos necesarios para que la librería funcione en paralelo se presentan al final del capítulo, ya que a nuestro modo de ver la parte más relevante para el lector del trabajo consiste en la forma de uso de la librería, así como en las múltiples opciones que nos permite. Uno de los principales motivos para realizar dicha extensión al entorno paralelo consiste en el uso que puede hacerse de dicha librería para realizar el análisis de especulación en Eden. Por este motivo, la sección principal del capítulo describe la forma de uso de dicha librería para la especulación, incluyendo explicaciones de las gráficas obtenidas para la resolución de un sistema de ecuaciones lineales.

En el Capítulo 8 se presenta un estudio exhaustivo de las posibles semánticas que podría tener el depurador en los entornos paralelos GpH y Eden. Entre ellas se presenta la semántica de la implementación vista en el Capítulo 7, se estudian sus características y se explican sus ventajas y desventajas. El estudio de estas semánticas se realiza formalmente y, al igual que en el Capítulo 6, se prueba su corrección con respecto a las semánticas sin observaciones y se presentan ejemplos del comportamiento semántico de varias expresiones. Las demostraciones formales se presentan en el apéndice.

Finalmente, en el Capítulo 9 presentamos las principales conclusiones de esta tesis, así como las principales líneas de trabajo futuro.

El trabajo concluye con un apéndice en el que se presentan las demostraciones de los teoremas de los Capítulos 6 y 8. Hemos decidido presentar las demostraciones en un apéndice para hacer más legible esta tesis. De esta forma el lector que sólo esté interesado en las innovaciones que aquí presentamos puede leer esta tesis obviando las demostraciones. Por su parte, el lector interesado en la parte formal sólo tendrá que pasar al apéndice para encontrar los detalles de las demostraciones.

Los resultados más relevantes de esta tesis han sido publicados en [ELR05, ELR06a, ERR07, ELRH07].

Capítulo 2

Semántica natural y máquinas abstractas

La pretensión de este capítulo es la de realizar un resumen tanto de la semántica de los lenguajes funcionales perezosos como de las máquinas abstractas que se han desarrollado a lo largo de los años para la evaluación de dichos lenguajes. Por esto, nos centraremos tanto en las semánticas como en las máquinas abstractas que posteriormente utilizaremos para el desarrollo de esta tesis. El trabajo realizado ha sido el de unificar la notación de las semánticas y las máquinas abstractas que aquí se presentan.

Inicialmente se desarrollaron varias máquinas abstractas: la máquina G [Aug84, Joh84, Pey87] (1984), la máquina TIM (1987) [FW87, PL92], la máquina de Krivine (1991) [Cur91] y la máquina STG (Spineless Tagless G Machine) (1992) [PS89, Pey92]. Estas primeras máquinas no prestaban demasiada atención a la corrección, sino solamente a la eficiencia. Posteriormente se desarrollo la semántica abstracta: Launchbury en 1993 definió una semántica operacional para la evaluación perezosa y desarrolló las pruebas de corrección y completitud con respecto a una semántica denotacional. Nosotros en este capítulo primero presentaremos las semánticas operacionales y posteriormente las máquinas abstractas. Con respecto a la semántica, presentaremos la semántica de Launchbury [Lau93] y la posterior modificación de esta por Sestoft [Ses97]. La principal modificación a nivel semántico de Sestoft consiste en la modificación del renombramiento que lo cambió de la regla de evaluación de la variable a la regla de evaluación de la expresión **letrec** para mantener la diferencia clara entre los punteros y las variables de programa.

Con respecto a las máquinas, en este capítulo presentaremos la máquina STG y las máquinas de Sestoft. Presentaremos la máquina STG porque la consideramos una de las más óptimas que actualmente se encuentra implementada en el compilador GHC (Glasgow Haskell Compiler). Además, presentaremos las máquinas de Sestoft ya que, aunque se desarrollaron posteriomente a la *STG*, son bastante más sencillas y didácticas.

Otra de las extensiones que nos parecen interesantes a estudiar y que posteriormente utilizaremos es la relativa a los valores primitivos. Para ello partiremos como base de [PL92].

2.1. Semántica natural

John Launchbury define en [Lau93] una semántica operacional natural de paso grande para la evaluación perezosa (reducción en orden normal hasta forma normal débil de cabeza y compartición de los argumentos). Inicialmente parte de un pequeño λ-cálculo normalizado, extendido con expresiones **let** recursivas[1] y define sobre dicho lenguaje una semántica operacional. Tras esta etapa comprueba la corrección y equivalencia de la semántica operacional con respecto a una semántica denotacional del lenguaje funcional perezoso.

Posteriormente extiende tanto la semántica operacional como el lenguaje con constructores saturados, expresiones **case**, números y operaciones primitivas. Pero no demuestra su corrección con respecto a la semántica denotacional.

Finalmente comenta extensiones tales como la recolección de basura y costes de computación, que en este capítulo no se presentarán.

En [Ses97] Peter Sestoft parte del mismo lenguaje funcional perezoso que Launchbury, así como de su semántica operacional. La semántica de Sestoft es una modificación de la de Launchbury. La principal modificación es el lugar donde se generan variables frescas, que pasa a ser la regla de evaluación del **letrec** y que añade un conjunto con la idea de comprobar localmente la frescura de las variables.

Posteriormente demuestra ciertas propiedades de su semántica revisada, entre ellas, que la frescura se puede comprobar de forma local en cada regla, que no hay captura de variables y que las variables libres y ligadas son conjuntos disjuntos en cualquier derivación.

En ese momento deriva varias máquinas abstractas y finalmente extiende tanto la semántica como las máquinas abstractas con constructores saturados, expresiones **case**, números y operaciones primitivas. Pero no demuestra que dicha extensión mantiene las propiedades. Es más, la propiedad de que la frescura se pueda comprobar de forma local en cada regla se pierde. Por tanto, como vimos en [EP01] se podría llegar a realizar alguna captura de variables. Por ese motivo, aquí presentaremos la corrección que realizamos sobre la semántica de Sestoft en [EP01].

2.1.1. Sintaxis

En la Figura 2.1 se puede encontrar el λ-cálculo utilizado tanto por Launchbury como por Sestoft. La notación $\overline{A_i}$ denota un vector $A_1,\ldots,A_n$ de valores subindexados y x e y se corresponden con variables. Este lenguaje es un λ-cálculo normalizado extendido con constructores y expresiones **case**, pero no con valores primitivos. Posteriormente, en el Capítulo 2.4 trataremos la extensión relativa a los valores primitivos, números enteros, etc.

Otro de los aspectos a resaltar es que este lenguaje sólo posee expresiones **let** recursivas, o mejor dicho, potencialmente recursivas; en este nivel semántico no existe ninguna diferencia en la evaluación de una expresión **let** no recursiva con respecto a la evaluación de la recursiva.

Es importante comentar que este lenguaje no pierde ninguna potencia expresiva con respecto al λ-cálculo tradicional, ya que cualquier expresión en este último puede ser transformada a una expresión en este λ-cálculo normalizado. Así, el proceso de normalización implica las siguientes transformaciones:

1. Todas las variables ligadas del programa (ya sean en las expresiones lambda, **let** o **case**) de la expresión inicial deben ser distintas.

[1] En nuestra sintaxis **letrec**, en su sintaxis **let**.

		-- Expresiones	
e	$\rightarrow$	x	-- variable
	$\mid$	$\lambda x.e$	-- lambda abstracción
	$\mid$	$e\ x$	-- aplicación
	$\mid$	**letrec** $\overline{x_i = e_i}$ **in** e	-- let recursivo
	$\mid$	$C\ \overline{x_i}$	-- aplicación saturada de constructor
	$\mid$	**case** e **of** $alts$	-- expresión case
		-- Alternativas	
$alts$	$\rightarrow$	$\overline{C_i\ \overline{x_{ij}} \mapsto e_i}$	-- alternativas
		-- Formas normales débiles de cabeza ($whnf$)	
w	$\rightarrow$	$C\ \overline{x_i}$	-- aplicación de constructores
	$\mid$	$\lambda\ x.e$	-- abstraccion lambda

Figura 2.1: λ-calculo de Launchbury

2. Toda aplicación debe realizarse a una variable y todo constructor debe estar saturado, es decir, aplicado a todos sus argumentos. Para ello, lo único necesario es añadir ligaduras **let** para aquellos argumentos que no sean variables.

   De esta forma, manteniendo los argumentos como variables, cada vez que se evalúe un argumento y se actualice su valor con su forma normal, su evaluación se mantiene para los cómputos restantes y de este modo, la siguiente vez que accedamos a ese argumento estará evaluado y utilizaremos directamente su forma normal.

Las formas normales (débiles de cabeza) de este λ-cálculo normalizado son las lambda abstracciones y los constructores saturados.

Las expresiones **letrec** son potencialmente recursivas, y todas las variables ligadas dentro de un **letrec** han de ser distintas. Finalmente, tal y como es estándar en el λ-cálculo, las aplicaciones se realizan a un único argumento.

2.1.2. Semántica operacional de Launchbury

A la hora de definir dicha semántica operacional, la única maquinaria necesaria es un *heap* explícito donde se mantienen las ligaduras. Un *heap* consiste en una función finita entre variables y expresiones, y puesto que es una función, no se permiten ligaduras duplicadas para la misma variable.

En la Figura 2.2 pueden verse las reglas semánticas. A lo largo de todo el trabajo el símbolo w será utilizado para denotar una forma normal. Un juicio de la forma $H : e \Downarrow L : w$ significa que la expresión e, con sus variables libres ligadas en el *heap* H, se reduce a una forma normal w y produce el *heap* final L. Si en un *heap* se añadiera una nueva ligadura para una variable ligada previamente, se reemplazaría la expresión asociada a esa variable por la nueva expresión. Sin embargo, esto no sucede ni en la semántica de Launchbury ni en la de Sestoft por lo que utilizaremos la notación $H \cup [x \mapsto e]$ para indicar que el *heap* está formado por una unión disjunta de H y $[x \mapsto e]$.

$$H : \lambda x.e \Downarrow H : \lambda x.e \qquad \textit{Lam}$$

$$H : C\,\overline{x_i} \Downarrow H : C\,\overline{x_i} \qquad \textit{Cons}$$

$$\frac{H : e \Downarrow K : \lambda y.e' \quad K : e'[x/y] \Downarrow L : w}{H : e\ x \Downarrow L : w} \qquad \textit{App}$$

$$\frac{H : e \Downarrow K : w}{H \cup [x \mapsto e] : x \Downarrow K \cup [x \mapsto w] : \hat{w}} \qquad \textit{Var}$$

$$\frac{H \cup [\overline{x_i \mapsto e_i}] : e \Downarrow K : w}{H : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \Downarrow K : w} \qquad \textit{Letrec}$$

$$\frac{H : e \Downarrow K : C_k\ \overline{x_j} \quad K : e_k[\overline{x_j/y_{kj}}] \Downarrow L : w}{H : \textbf{case } e \textbf{ of } \overline{C_i\ \overline{y_{ij}} \to e_i} \Downarrow L : w} \qquad \textit{Case}$$

Figura 2.2: Semántica natural de Launchbury

En Launchbury, la notación $\hat{w}$ significa que en la expresión w hemos reemplazado todas las variables *ligadas* por nombres frescos.

Decimos que $H : e \Downarrow L : w$ es una derivación exitosa si se puede probar usando las reglas. La prueba de una derivación puede fallar por varios motivos, uno de ellos puede ser que el programa no termine y se quede en un bucle infinito sin llegar a derivar una forma normal. Otra posibilidad es que no haya regla para aplicar y que no se pueda alcanzar una forma normal. Entre estas últimas situaciones cabe destacar el caso en que no hay regla para aplicar debido a que se referencia una clausura que actualmente está bajo evaluación. Entonces diremos que la evaluación ha entrado en un *agujero negro*. Esto podría suceder en la regla *Var* cuando aparece una referencia a la variable x mientras estamos reduciendo la expresión e y antes de alcanzar una forma normal. Como el *heap* H no contiene ninguna ligadura para la variable x, no se puede aplicar ninguna regla y, por tanto, la derivación no puede completarse. Otra situación interesante de no finalización sería que el discriminante de una expresión **case** hubiera reducido a un constructor que no se encontrara dentro de las alternativas de dicha expresión.

El principal teorema en [Lau93] es que la semántica operacional es correcta y completa con respecto a la semántica denotacional para un lenguaje no estricto, esto significa que si e es una expresión cerrada, entonces $[\![e]\!]\ \rho_0 = v \neq \perp$, es decir, v es el valor de su semántica operacional bajo el entorno ρ_0, si y sólo si existe L y w tales que $\{\ \} : e \Downarrow L : w$ y $[\![w]\!]\ \rho_L = v$, siendo ρ_0 el entorno vacío y ρ_L el entorno que se obtiene a partir del *heap* L donde se halla la definición de las variables libres de la expresión w.

2.1.3. Semántica de Sestoft

Como comentamos anteriormente, Sestoft parte del mismo lenguaje e introduce en [Ses97] dos cambios importantes en la semántica operacional de la Figura 2.3:

$$H : \lambda x.e \ \Downarrow_{A,C} \ H : \lambda x.e \qquad \textit{Lam}$$

$$H : C\,\overline{p_i} \ \Downarrow_{A,C} \ H : C\,\overline{p_i} \qquad \textit{Cons}$$

$$\frac{H : e \ \Downarrow_{A,C} \ K : \lambda x.e' \quad K : e'[p/x] \ \Downarrow_{A,C} \ L : w}{H : e\ p \ \Downarrow_{A,C} \ L : w} \qquad \textit{App}$$

$$\frac{H : e \ \Downarrow_{A \cup \{p\},C} \ K : w}{H \cup [p \mapsto e] : p \ \Downarrow_{A,C} \ K \cup [p \mapsto w] : w} \qquad \textit{Var}$$

$$\frac{H \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \ \Downarrow_{A,C} \ K : w}{H : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \ \Downarrow_{A,C} \ K : w} \text{ donde } \overline{p_i} \text{ frescas} \qquad \textit{Letrec}$$

$$\frac{H : e \ \Downarrow_{A,C \cup \{\overline{C_i \ \overline{y_{ij}} \to e_i}\}} \ K : C_k \ \overline{p_j} \quad K : e_k[\overline{p_j/y_{kj}}] \ \Downarrow_{A,C} \ L : w}{H : \textbf{case } e \textbf{ of } \overline{C_i \ \overline{y_{ij}} \to e_i} \ \Downarrow_{A,C} \ L : w} \qquad \textit{Case}$$

Figura 2.3: Semántica natural de Sestoft

1. Cambia el renombramiento de variables de la regla *Var* a la regla *Letrec*.

2. Extiende los juicios con un nuevo conjunto A de *variables bajo evaluación* para poder mantener la propiedad de frescura de forma local en cada regla. Por tanto, facilita poder generar nombres frescos de forma adecuada en la regla *Letrec*.

   Como se vio en [EP01], únicamente con este conjunto no se mantiene la propiedad de frescura de forma local cuando en el lenguaje tenemos expresiones **case**. Por tanto, presentaremos la semántica con dos conjuntos, A y C, donde C se corresponderá con el conjunto de alternativas pendientes de aplicar de las expresiones **case**. Esto sólo es necesario para la definición de frescura.

La primera modificación se debe al intento de acercar un poco más la semántica a la implementación de las máquinas abstractas. En una implementación usual, las variables frescas (es decir, *los punteros*) se crean cuando introducimos clausuras en el *heap* en la regla *Letrec*. Esta aproximación es también más "económica" que la consistente en renombrar todas las variables ligadas de una lambda abstracción, ya que ahora sólo se necesita renombrar las variables que han pasado de ligadas a libres (es decir, cuando pasa de ser una variable de programa a ser un *puntero*). Gracias a este cambio se consigue una diferencia clara entre las variables libres (punteros) y ligadas del programa. Además, se evita la necesidad de que todas las variables ligadas de la expresión inicial tengan distintos nombres.

La segunda modificación hace más precisa la definición de *frescura*: una variable es fresca en un juicio $H \ : e \ \Downarrow_{A,C} \ L : w$ si no pertenece ni a *var* H ni a A, ni a *var* e y no se encuentra en *var* C. Por *var* J entendemos todas las variables que se encuentran en J.

Las reglas modificadas pueden verse en la Figura 2.3. En la regla *Letrec*, la frescura ha de entenderse con respecto al juicio $H : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \ \Downarrow_{A,C} \ K : w$. La notación $\hat{e}$, en Sestoft,

significa que en la expresión e hemos reemplazado las variables x_i por los punteros p_i, es decir, $\hat{e} = e[\overline{p_i/x_i}]$.

El conjunto A contiene las direcciones de las clausuras actualmente en evaluación, por lo tanto, cuando vamos a evaluar una variable, metemos en el conjunto A la variable y la quitamos del *heap* (véase la regla *Var*), de tal manera que en el *heap* no mantenemos las variables que se encuentran en evaluación. Es por ese motivo, que para mantener la frescura local Sestoft almacena las variables en dicho conjunto. No obstante, a nivel semántico, la eliminación de la clausura del *heap* en la regla *Var* sólo nos lleva a que el cómputo de una expresión que trate de entrar en una variable bajo evaluación, en vez de ciclar, pare sin posibilidad alguna de continuar con la reducción. Por tanto, a nivel semántico sería equivalente mantener las clausuras bajo evaluación en el *heap*: tanto si cicla como si se para sin haber alcanzado *whnf*, diremos que dicha expresión no posee semántica.

La diferencia entre las nuevas reglas (ver Figura 2.3) y las dadas en la Figura 2.2 es el lugar donde sucede el renombramiento. En consecuencia, lo único que es necesario probar para demostrar la equivalencia entre ambas semánticas es que no existe ninguna captura de variables en la sustitución $e'[x/y]$ en la regla *App*, y que no hay ningún intento de ligar en el *heap* ninguna variable que estuviera ya ligada en K, en la regla *Var*. Esto lo demuestra Sestoft con la siguiente proposición:

Proposición 2.1 (Sestoft) [2] *Si* $H_0 : e_0 \Downarrow_{\{\},\{\}} L : w_0$ *es una derivación, en todos los juicios de dicha derivación* $H : e \Downarrow_{A,C} L : w$ *las siguientes propiedades se mantienen:*

1. $(dom\ H) \cap A = \varnothing$.
2. $fv\ e \subseteq (dom\ H) \cup A$.
3. $bv\ e \cap ((dom\ H) \cup A) = \varnothing$
4. *Para toda ligadura* $[p \mapsto e'] \in H$ *se cumple que* $fv\ e' \subseteq (dom\ H) \cup A$ *y* $bv\ e' \cap ((dom\ H) \uplus A) = \varnothing$

siendo fv e el conjunto de variables libres de la expresión e y bv e el conjunto de variables ligadas de la expresión e.

En otras palabras, en cada juicio del árbol de derivación existe una clara distinción entre las variables libres y las variables ligadas: las primeras se encuentran ligadas en el correspondiente *heap* o bajo evaluación, es decir, en A. Mientras que las segundas son las variables de programa que se encontraban en la expresión original escrita por el programador. Utilizaremos el término *punteros* para referirnos a las variables frescas creadas de forma dinámica y el termino *variables de programa* para referirnos a las variables de programa. Utilizaremos los nombres de variables $p, q, \ldots$ para denotar los punteros y $x, y, \ldots$ para denotar las variables de programa.

Sestoft demuestra que su semántica y la de Launchbury llegan a las mismas deducciones y, por tanto, son equivalentes.

[2] Esta proposición es una mezcla de la Definición 1, el Lema 1 y la Proposición 1 de Sestoft.

	Heap	Control	Pila	Regla
	H	$e\ p$	S	*app1*
$\Longrightarrow$	H	e	$p : S$	
	H	$\lambda y.e$	$p : S$	*app2*
$\Longrightarrow$	H	$e[p/y]$	S	
	$H \cup [p \mapsto e']$	p	S	*var1*
$\Longrightarrow$	H	e'	$\#p : S$	
	H	$\lambda y.e$	$\#p : S$	*var2*
$\Longrightarrow$	$H \cup [p \mapsto \lambda y.e]$	$\lambda y.e$	S	
	H	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	S	*letrec*(1)
$\Longrightarrow$	$H \cup [\overline{p_i \mapsto \hat{e}_i}]$	$\hat{e}$	S	
	H	**case** e **of** *alts*	S	*case1*
$\Longrightarrow$	H	e	*alts* $: S$	
	H	$C_k\ \overline{p_i}$	*alts* $: S$	*case2*(2)
$\Longrightarrow$	H	$e_k[\overline{p_i/y_{ki}}]$	S	
	H	$C_k\ \overline{p_i}$	$\#p : S$	*var3*
$\Longrightarrow$	$H \cup [p \mapsto C_k\ \overline{p_i}]$	$C_k\ \overline{p_i}$	S	

(1) $\hat{e} = [\overline{p_i/x_i}]$ y $\overline{p_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = e_i}\}$ **in** e y S.
(2) e_k se corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en *alts*

Figura 2.4: Máquina abstracta *Mark-1*

2.2. Máquinas de Sestoft

A partir de su semántica operacional, Sestoft introduce en diferentes etapas varias máquinas abstractas *Mark-1*, *Mark-2* y *Mark-3*. Estas máquinas van de más cercanas a la semántica operacional hacia más cercanas a la implementación real. A continuación las describiremos brevemente.

2.2.1. La máquina abstracta *Mark-1*

La primera máquina abstracta es la *Mark-1* (véase Figura 2.4), que está muy cercana a la semántica y está basada en la de Krivine [Cur91], que es una máquina para la llamada por nombre, modificada para la evaluación perezosa y extendida a los constructores y expresiones **case**. Cada dos líneas separadas por $\Rightarrow$ representan una posible transición de la máquina. El único añadido necesario para esta máquina abstracta es una pila S en la que se almacenan los argumentos por aplicar, las marcas de actualización y las alternativas (reglas *app1*, *var1* y *case1* respectivamente), es decir, la pila almacena el contexto de tal forma que cuando terminamos la evaluación de un juicio en las reglas *App* y *Case* tenemos que acceder a la cima de la pila donde

se almacenó el contexto y proseguir la evaluación del segundo juicio con lo que nos indique la cima de la pila. En ella se realizan sustituciones explícitas: en la regla *letrec*, en la regla *app2*; cuando aplicamos una lambda abstracción a su argumento realizamos la sustitución $e[p/y]$; y en la regla *case2*, cuando realizamos el ajuste de patrones realizamos la sustitución $e_k[\overline{p_i/y_{ki}}]$. La notación $x : xs$ representa la concatenación por la izquierda de un elemento x a una lista xs.

Las reglas semánticas *Var*, *App* y *Case* han sido divididas en las correspondientes reglas *var1*, *var2* y *var3*, *app1* y *app2* y *case1* y *case2*. La división de la regla *Var* se debe a que la actualización, reglas *var2* y *var3*, ha de realizarse después de la evaluación de la expresión, regla *var1*. Sin embargo la división de las reglas *App* y *Case* se debe al cómputo correspondiente a la ejecución de cada juicio en la parte superior de las reglas semánticas correspondientes.

En las reglas *app2*, *var2*, *var3* y *case2* es la pila la que toma el control y según lo que se halle en la cima se realiza un cómputo distinto. Así sucede cuando llegamos a una forma normal. Si ésta es una lambda abstracción y en la pila se encuentra una marca de actualización (regla *var2*), se almacena dicha lambda abstracción en el *heap* en la dirección correspondiente, y si en la pila se encuentra un argumento por aplicar (regla *app2*), se aplica la lambda abstracción a dicho argumento. Si la expresión de control es una construcción y en la pila se encuentra una marca de actualización (regla *var3*), se almacena dicho constructor en el *heap* en la dirección correspondiente: y si en la pila se encuentran unas alternativas (regla *case2*), se realiza el encaje de patrones y se evalúa la alternativa correspondiente a dicho constructor.

Teorema 2.1 *Sestoft demostró que el comportamiento de esta máquina es exactamente el mismo que el de la semántica, en el sentido de que para toda expresión cerrada* e, $\{\,\} \; e \Downarrow_{\{\,\},\{\,\}} K\, w$ *es derivable si y sólo si* $(\{\,\}, e, [\,]) \Rightarrow^* (H, w, [\,])$ *donde* $\Rightarrow^*$ *significa una evaluación de cero o más transiciones* $\Rightarrow$ *de la máquina Mark-1. La demostración hace uso de la noción de evolución equilibrada de la máquina.*

2.2.2. La máquina abstracta *Mark-2*

La máquina abstracta *Mark-2*, que se muestra en la Figura 2.5, se aleja algo más de la semántica. Elimina las sustituciones explícitas de variables, ya que éstas no son fáciles de realizar en tiempo de ejecución. Para ello necesita un entorno E que es una función que asocia a variables del programa un puntero. Ahora cada expresión, tanto en el *heap* como la expresión de control, lleva asociado un entorno que liga sus variables libres con las direcciones en el *heap* donde se encuentran almacenadas las clausuras, pares del tipo (e, E), correspondientes. En lugar de realizar la sustitución en las reglas *app2* y *case2*, lo que se hace es añadir en el entorno la ligadura $y \mapsto p$ o las ligaduras $\overline{y_{ki} \mapsto p_i}$, de tal manera que ahora cuando nos encontramos con una variable, regla *var1*, consultamos en el entorno la dirección en la que se encuentra en el *heap* y accedemos a ella. Cuando se realiza una actualización del *heap* ahora es necesario almacenar la expresión junto con el entorno asociado a dicha expresión, y cuando se accede al *heap* para extraer una clausura también es necesario sacar con ella su entorno asociado.

Teorema 2.2 *Sestoft dejó indicada la demostración de que el comportamiento de esta máquina y el de la máquina Mark-1 es el mismo y, por tanto, también es correcta y completa con respecto a la semántica.*

Hemos utilizado la misma notación para el entorno que la que usamos para el *heap*, con el mismo significado. Así, $E \cup \{x \mapsto p\}$ significa que el entorno $E \cup \{x \mapsto p\}$ es la unión de E con

	Heap	Control	Entorno	Pila	Regla
	H	$e\ x$	$E\{x \mapsto p\}$	S	*app1*
$\Longrightarrow$	H	e	E	$p : S$	
	H	$\lambda y.e$	E	$p : S$	*app2*
$\Longrightarrow$	H	e	$E \cup \{y \mapsto p\}$	S	
	$H \cup [p \mapsto (e', E_1)]$	x	$E\{x \mapsto p\}$	S	*var1*
$\Longrightarrow$	H	e'	E_1	$\#p : S$	
	H	$\lambda y.e$	E	$\#p : S$	*var2*
$\Longrightarrow$	$H \cup [p \mapsto (\lambda y.e, E)]$	$\lambda y.e$	E	S	
	H	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	E	S	*letrec*(^{1})
$\Longrightarrow$	$H \cup [\overline{p_i \mapsto (e_i, E_1)}]$	e	E_1	S	
	H	**case** e **of** *alts*	E	S	*case1*
$\Longrightarrow$	H	e	E	$(alts, E) : S$	
	H	$C_k\ \overline{x_i}$	$E\{\overline{x_i \mapsto p_i}\}$	$(alts, E_1) : S$	*case2*(^{2})
$\Longrightarrow$	H	e_k	$E_1 \cup \{\overline{y_{ki} \mapsto p_i}\}$	S	
	H	$C_k\ \overline{x_i}$	E	$\#p : S$	*var3*
$\Longrightarrow$	$H \cup [p \mapsto (C_k\ \overline{x_i}, E)]$	$C_k\ \overline{x_i}$	E	S	

(^{1}) $\hat{e} = [\overline{p_i/x_i}]$, $\overline{p_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = e_i}\}$ **in** e y S y $E_1 = E \cup \{\overline{x_i \mapsto p_i}\}$.

(^{2}) e_k se corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en *alts*

Figura 2.5: Máquina abstracta *Mark-2*

la ligadura $x \mapsto p$. Si ésta existiera se sobreescribría. Si partimos de una expresión cerrada con todas las variables ligadas con nombres distintos, en ningún entorno se podría ligar una variable que ya estuviera ligada previamente. Además, se ha añadido la notación $E\{x \mapsto p\}$ que resalta que $(x \mapsto p) \in E$.

2.2.3. La máquina abstracta *Mark-3*

Posteriormente, Sestoft introduce la máquina abstracta *Mark-3*, ver Figura 2.6, donde las variables han sido sustituidas por los denominados índices de De Bruijn correspondientes. Para más detalles véase [Bru72, Bru78, ACCL90, NW98, BP99, Enc01].

La principal diferencia ahora es la representación del entorno E, que ha pasado de ser una función a representarse con una lista de variables. Ahora el acceso al puntero relativo a una variable se realiza con el índice de De Bruijn. El índice de De Bruijn es un número entero positivo que nos indica el número de variables ligadas existentes entre la ligadura de dicha variable, es decir, su aparición de definición, y las apariciones de uso de la variable. Para realizar este cálculo se tienen en cuenta tanto las variables de las lambda abstracciones como las variables

	Heap	Control	Entorno	Pila	Regla
	H	$e\ n$	$\overline{p_i}^n : E$	S	*app1*
$\Longrightarrow$	H	e	$\overline{p_i}^n : E$	$p_n : S$	
	H	$\lambda.e$	E	$p : S$	*app2*
$\Longrightarrow$	H	e	$p : E$	S	
	$H \cup [p_n \mapsto (e', E_1)]$	n	$\overline{p_i}^n : E$	S	*var1*
$\Longrightarrow$	H	e'	E_1	$\#p_n : S$	
	H	$\lambda.e$	E	$\#p : S$	*var2*
$\Longrightarrow$	$H \cup [p \mapsto (\lambda.e, E)]$	$\lambda.e$	E	S	
	H	**letrec** $\{\overline{e_i}\}$ **in** e	E	S	*letrec*(^{1})
$\Longrightarrow$	$H \cup [\overline{p_i \mapsto (e_i, E_1)}]$	e	E_1	S	
	H	**case** e **of** *alts*	E	S	*case1*
$\Longrightarrow$	H	e	E	$(alts, E) : S$	
	H	$C_k\ \overline{u_i}^n$	E	$(alts, E_1) : S$	*case2*(^{2})
$\Longrightarrow$	H	e_k	$E[u_n] : \ldots : E[u_1] : E_1$	S	
	H	$C\ \overline{u_i}$	E	$\#p : S$	*var3*
$\Longrightarrow$	$H \cup [p \mapsto (C\ \overline{u_i}, E)]$	$C\ \overline{u_i}$	E	S	

(^{1}) $\hat{e} = [\overline{p_i/x_i}]$, $\overline{p_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = e_i}\}$ **in** e y S y $E_1 = \overline{p_i} : E$.

(^{2}) e_k se corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en *alts*.

Figura 2.6: Máquina abstracta *Mark-3*

que aparecen en las expresiones **case** y **letrec**. Cuando vamos a acceder a una clausura del *heap* lo hacemos con un entero en la expresión de control, denotado por n en la máquina. Lo único que tenemos que hacer es consultar en el entorno E la posición n y ésta nos indica la dirección del *heap* donde se encuentra ligada dicha variable.

Esto conlleva un ahorro en ejecución tanto del tiempo como del espacio, ya que ahora no es necesario almacenar las variables del programa en los entornos sino sólo los punteros manteniendo un orden, que es el orden temporal en el cual se fueron creando dichas variables libres, es decir, de esta manera los entornos sólo tienen que almacenar los punteros, ya que las variables son valores enteros. Por tanto, la notación $E[u_i]$ es el valor existente en el entorno E en la posición u_i.

Aunque esta máquina se presenta aquí, se hace únicamente por completitud, ya que no se utilizará para el desarrollo de este trabajo, al considerar que la sustitución de variables por números enturbia los detalles de la evaluación y sólo se corresponde con una optimización que se puede desarrollar en el proceso de compilación.

2.2.4. Poda de entornos

Sestoft introdujo en sus máquinas varias mejoras. Una de las que consideramos importantes y tendremos en cuenta en esta tesis, se corresponde con la *poda de los entornos*. Esta poda se introdujo en las máquinas *Mark-2* y *Mark-3* con la pretensión de acercarlas a la idea de sustitución que aparece tanto en la semántica como en la máquina *Mark-1*. De esta forma se consigue que se acerquen a la implementación real donde se realiza la *poda de los entornos*. Idealmente, tanto el entorno E de la máquina, como los entornos almacenados en las clausuras del *heap* no deberían vincular variables "superfluas", es decir, variables que no se encuentren libres en la expresión de control o en la clausura correspondiente. Sin embargo, esta tarea resulta costosa si esa poda se realiza cada vez que se evalúa un paso la expresión de control. Pero, por otro lado, resulta muy ineficiente almacenar en cada clausura todo el entorno. Por eso, el compromiso al que se suele llegar es no podar el entorno de la expresión de control, ya que en ciertos momentos éste desaparecerá (regla *Var*), pero sí podar los entornos que se almacenan en las clausuras y los que se almacenan en la pila.

Por ese motivo, nosotros utilizaremos las notaciones $H \cup [p \mapsto (e, E|)]$ y $(alts, E|)$ que significan quedarse únicamente con los siguientes nombres de variables del entorno: *fv e* o *fv alts* respectivamente, es decir, siguiendo la notación de Sestoft: $E|^{fv\ e}$ o $E|^{fv\ alts}$ respectivamente.

Nos gustaría destacar que esta modificación no afecta para nada a la evaluación de cualquier expresión, ya que desde la expresión correspondiente sólo se pueden alcanzar sus variables libres.

2.3. La máquina abstracta *STG*

Peyton Jones en [Pey92] describe en detalle el comportamiento de la máquina STG que está implementada en el compilador GHC [PHH+93]. Este trabajo se realizó posteriormente al compilador GHC, por lo que en él se modelan muchas de las optimizaciones y de detalles de bajo nivel que posee dicho compilador, tales como ligaduras principales, formatos de las clausuras, punteros, código C, etc.

Para describir el comportamiento de dicha máquina, primero presenta un lenguaje (STGL) que, aún siendo un λ-cálculo, posee una sintaxis diferente. En dicho lenguaje se encuentran optimizaciones producidas por varios análisis: el de estrictez, el de uso, etc. Este lenguaje de partida posee una expresión **let** y una expresión **letrec** que evalúan de forma diferente. Por otro lado, las aplicaciones y las λ-abstracciones poseen varios argumentos, con el fin de optimizar su cómputo. Además, incluye de partida los valores primitivos, las operaciones sobre dichos valores y las expresiones **case** sobre valores primitivos.

Tras esto, Peyton Jones presenta una máquina abstracta con varios detalles de bajo nivel, un puntero que apunta a la clausura bajo evaluación, tres pilas diferentes (una para los argumentos, otra para las actualizaciones y otra para los retornos) y un entorno global, para optimizar el acceso a las clausuras que se encuentran a nivel principal y almacenar menos direcciones en las clausuras.

Finalmente, explica diversas optimizaciones y da una idea las diversas posibilidades de traducción de la máquina a código C, así como de la traducción adoptada en el compilador GHC.

Como ya se comentó, dicha máquina se encuentra implementada dentro del compilador GHC. Por tanto, tanto el lenguaje como la máquina es de más bajo nivel que los presentados anteriormente. Por esta razón, nosotros presentaremos aquí una versión simplificada pero equivalente de ambos. Para más detalles véase [Pey92, Enc01, EP01, ELR06b].

		-- Expresiones	
e	$\rightarrow$	$x\ \overline{atom_i}^n$	-- aplicación
	$\mid$	$op\ \overline{atom_i}^n$	-- operador primitivo saturado
	$\mid$	$atom$	-- variable o literal
	$\mid$	**let** $\overline{x_i = be_i}^n$ **in** e	-- let
	$\mid$	**letrec** $\overline{x_i = be_i}$ **in** e	-- let recursivo
	$\mid$	**case** e **of** $alts$	-- expresión case
		-- Ligaduras	
be	$\rightarrow$	w	-- formas normales débiles de cabeza
	$\mid$	e	-- expresión
		-- Formas normales débiles de cabeza ($whnf$)	
w	$\rightarrow$	$C\ \overline{atom_i}$	-- aplicación de constructores
	$\mid$	$\lambda\ \overline{x_i}.e$	-- abstracción lambda
		-- Alternativas	
$alts$	$\rightarrow$	$\overline{C_i\ \overline{x_j}^{k_i} \mapsto e_i}^k.default$	-- alternativas algebraicas
	$\mid$	$\overline{prim_i \mapsto e_i}^k.default$	-- alternativas primitivas
$default$	$\rightarrow$	**otherwise** $\mapsto e$	
	$\mid$	$y \mapsto e$	
	$\mid$	ϕ	
		-- Átomos	
$atom$	$\rightarrow$	$x,\ y,\ p,\ q$	-- variables
	$\mid$	$prim$	-- valores primitivos
		-- Primitivos	
$prim$	$\rightarrow$	$int(1\#,\ 2\#,\ \ldots)$	-- enteros primitivos
	$\mid$	$\ldots$	-- otros
		-- Operaciones sobre primitivos	
op	$\rightarrow$	$+\#$	-- suma
	$\mid$	$\ldots$	-- otras

Figura 2.7: Lenguaje STGL

2.3.1. Lenguaje STG (STGL)

El lenguaje presentado en la Figura 2.7 se corresponde con el lenguaje Core simplificado que la máquina *STG* evalúa. El compilador GHC transforma el código Haskell a este lenguaje. Para ello, asumiremos que n, k y m son enteros tales que $n > 0$, $k \geq 0$ y m se corresponde con un entero desencapsulado. Utilizaremos la notación $\overline{x_i}^n$ para representar n variables.

Como se puede ver, STGL es un λ-calculo normalizado, extendido con **let**, **letrec**, aplicaciones de constructores, expresiones **case** y enteros desencapsulados. El proceso de normalización fuerza a que la aplicación de constructores sea saturada y a que todas las aplicaciones se realicen sobre variables o valores desencapsulados. Las formas normales débiles de cabeza se corresponden con λ-abstracciones, constructores o valores primitivos.

Las principales diferencias de este lenguaje con respecto al de Launchbury son las siguientes:

- No se permiten aplicaciones de expresiones a variables. Todas las aplicaciones son de variable a variables. Por tanto, el proceso de normalización aún necesitará añadir ligaduras para todas las expresiones aplicadas a variables.
- Las aplicaciones son de una variable a varias, y no de una a una como sucede en el lenguaje de Launchbury (ver Figura 2.1).
- Sólo se admiten lambda abstracciones en las ligaduras de los **letrec**. (En adelante utilizaremos el término *lambda forma* para designar indistintamente a una lambda abstracción y a una expresión).
- Las expresiones **case** tienen una alternativa por defecto. Esta alternativa puede ser de dos tipos: la alternativa **otherwise**, que no almacena el valor resultante de la evaluación del discriminante del **case** y la alternativa con variable y, que almacena el valor del discriminante y lo asocia a la variable y.
- Existe una expresión **let** no recursiva.
- En este lenguaje tenemos valores desencapsulados, operaciones primitivas y expresiones **case** sobre valores desencapsulados. Nótese que no es posible crear una ligadura para un valor desencapsulado.

2.3.2. La máquina *STG*

Como hemos comentado previamente, la máquina original STG que aparecía en [Pey92] tenía tres pilas y sacaba los constructores a la expresión de control. Posteriormente sufrió varias modificaciones, sólo documentadas en el código del compilador. Entre las más importantes caben destacar las siguientes:

1. Los constructores, al igual que las λ-abstracciones, nunca salen a la expresión de control. Por tanto, las formas normales se corresponden con variables apuntando a constructores o λ-abstracciones.
2. Las tres pilas han sido colapsadas en una única pila. En [EP01] desarrollamos una máquina con una única pila y vimos su equivalencia con la máquina original STG.
3. Recientemente se ha modificado el modelo del cómputo de las aplicaciones, pasando del modelo original *push/enter* al modelo *eval/apply* (para más detalle véase [MP06, MP04]). Como comenta Peyton Jones, esta modificación sólo se ha realizado para que sea más sencilla la compilación, pues tanto en cuestiones de eficiencia como de equivalencia llega a los mismos resultados. Dicha modificación ha sido estudiada en detalle y se ha formalizado todo el proceso de compilación en un artículo que se encuentra pendiente de revisión y cuyo borrador se encuentra en [EP08].

	Heap	Control	Entorno	Pila	Regla
	H	**let** $\{\overline{x_i = be_i}\}$ **in** e	E	S	$let(^1)$
$\Rightarrow$	$H \cup [\overline{q_i \mapsto (be_i, E\|)}]$	e	E_1	S	
	H	**letrec** $\{\overline{x_i = be_i}\}$ **in** e	E	S	$letrec(^2)$
$\Rightarrow$	$H \cup [\overline{q_i \mapsto (be_i, E_1\|)}]$	e	E_1	S	
	H	**case** e **of** $alts$	E	S	$case1$
$\Rightarrow$	H	e	E	$(alts, E\|) : S$	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts, E_1) : S$	$case2(^3)$
$\Rightarrow$	H	e_k	$E_1 \cup \{\overline{y_{ki} \mapsto p_i}\}$	S	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.\textbf{otherwise} -> e, E_1) : S$	$case2d(^4)$
$\Rightarrow$	H	e	E_1	S	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.y -> e, E_1) : S$	$case2v(^4)$
$\Rightarrow$	H	e	$E_1 \cup \{y \mapsto q\}$	S	
	H	$x\ \overline{x_i}^n$	$E\{\overline{x_i \mapsto p_i}^n\}$	S	$app1$
$\Rightarrow$	H	x	E	$\overline{p_i}^n : S$	
	$H[q \mapsto (\lambda\overline{x_i}^n.e, E_1)]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^n : S$	$app2$
$\Rightarrow$	H	e	$E_1 \cup \{\overline{x_i \mapsto p_i}^n\}$	S	
	$H \cup [p \mapsto (e, E_1)]$	x	$E\{x \mapsto p\}$	S	$var1$
$\Rightarrow$	H	e	E_1	$\#p : S$	
	$H[p \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^k : \#q : S$	$var2(^5)$
$\Rightarrow$	$H \cup [q \mapsto (x\ \overline{x_i}^k, E_2\|)]$	x	E	$\overline{p_i}^k : S$	
	$H[q \mapsto (C_k\ \overline{x_i}, E_1)]$	x	$E\{x \mapsto q\}$	$\#p : S$	$var3$
$\Rightarrow$	$H \cup [p \mapsto (C_k\ \overline{x_i}, E_1)]$	x	$\{x \mapsto p\}$	S	
	H	x	$E\{x \mapsto k\}$	S	$prim\#$
$\Rightarrow$	H	k	$\{\}$	S	
	H	k	E	$(alts, E_1) : S$	$case2\#(^3)$
$\Rightarrow$	H	e_k	E_1	S	
	H	k	E	$(alts.\textbf{otherwise} -> e, E_1) : S$	$case2d\#(^4)$
$\Rightarrow$	H	e	E_1	S	
	H	k	E	$(alts.y -> e, E_1) : S$	$case2v\#(^4)$
$\Rightarrow$	H	e	$E_1 \cup \{y \mapsto k\}$	S	
	H	$op\ x_1\ x_2$	E	S	$op\#(^6)$
$\Rightarrow$	H	$m_1 op\ m_2$	$\{\}$	S	

(^{1}) $\overline{q_i}$ son distintas y frescas con respecto a H, **let** $\{\overline{x_i = be_i}\}$ **in** e y S. $E_1 = E \cup \{\overline{x_i \mapsto q_i}\}$
(^{2}) $\overline{q_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = be_i}\}$ **in** e y S.$E_1 = E \cup \{\overline{x_i \mapsto q_i}\}$
(^{3}) e_k se corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en $alts$
(^{4}) C_k no aparece en $alts$
(^{5}) $E_2 = \{x \mapsto p, \overline{x_i \mapsto p_i}^k\}$
(^{6}) $m_i =$ si $isInt(x_i)$ entonces x_i sino $E(x_i)$

Figura 2.8: La máquina abstracta STG

Las dos primeras modificaciones han sido consideradas para el desarrollo de esta tesis y sobre todo para el desarrollo del Capítulo 5.

La diferencia entre el modelo *eval/apply* y el modelo *push/enter* consiste en el modo en que se realizan las aplicaciones. En el modelo *push/enter* se depositan los argumentos en la pila y se entra a evaluar la clausura correspondiente (posiblemente una λ-abstracción), sin preocuparse si en la pila se encuentran los argumentos necesarios para que se pueda llevar a cabo la reducción de dicha función. Mientras que en el modelo *eval/apply* antes de entrar a evaluar la función hay que asegurarse que la pila contiene todos los argumentos necesarios para su reducción. Este tipo de cambio de evaluación no influye sustancialmente sobre la reducción de las marcas de observación. Además su inclusión haría bastante más complicada la comprensión de la semántica y de las máquinas tanto de Sestoft, como de GpH y Eden, cuyo modelo está basado en el modelo *push/enter* y a nivel semántico no supone ninguna diferencia. Es, por estos motivos, por los que hemos decidido no incluir dicha modificación en esta tesis.

La máquina abstracta *STG* presentada aquí ha sido adaptada a la notación de las máquinas de Sestoft. Para ello se han eliminado varios detalles no relevantes de la máquina *STG*, tales como el entorno global, el puntero a la clausura bajo evaluación y se han colapsado las tres pilas en una. Se utiliza la misma notación que en las máquinas anteriores y se ha añadido la notación $H[p \mapsto e]$ para resaltar que $(p \mapsto e) \in H$. También han sufrido cambio los nombres de las reglas, acercándolas a los nombres de reglas de Sestoft.

A continuación pasaremos a presentar las principales diferencias de esta máquina con respecto a la máquina *Mark-2* de Sestoft, que es de las tres máquinas, la que más se acerca a esta. Las principales diferencias son las siguientes:

1. Una de las diferencias más interesantes radica en que las aplicaciones se realizan a varias variables a la vez. Véanse las reglas *app1* y *app2*.

   Esto provoca que las actualizaciones con λ-abstraciones parcialmente aplicadas, regla *var2*, se realicen a través del siguiente tipo de clausura $(x\ \overline{x_i}^k, E)$ que no es más que una λ-abstracción aplicada parcialmente, *una aplicación parcial*. Téngase en cuenta que $k \geq 0$ y que en caso de que k fuera 0 se podría optimizar la regla y copiar directamente la λ-abstracción original en la clausura a actualizar. Esto duplicaría la regla. Por ese motivo hemos decidido mostrarla aquí de una forma compacta. Otro detalle a destacar es que debido a que las λ-abstracciones se aplican globalmente es necesario mirar si la cima de la pila contiene argumentos suficientes o si existe una marca de actualización entre medio de ellos.

2. Como ya se ha comentado, tanto los constructores como las λ-abstracciones no salen a la expresión de control. Por lo que las formas normales de esta máquina tienen la siguiente forma: $(H[p \mapsto w], x, E\{x \mapsto p\}, S)$.

   Esto provoca pequeñas modificaciones sobre las reglas *case2*, *var2* y *var3*, ya que ahora necesitan consultar el *heap* para realizar el encaje de patrones o la actualización.

3. En esta máquina abstracta, todos los entornos almacenados tanto en la pila como en el *heap*, es decir, los asociados a las alternativas y los de cada clausura respectivamente, son podados para que sólo contengan las variables libres de la expresión asociada.

4. Puesto que las expresiones **case** pueden contener una alternativa por defecto aparecen dos nuevas reglas para tratar dicha alternativa por defecto, las reglas *case2d* y *case2v*.

5. La máquina incluye valores primitivos desencapsulados, operaciones sobre dichos valores primitivos y expresiones **case** sobre dichos valores primitivos. Las reglas que trabajan sobre ellos son las reglas *prim#*, *case2#*, *case2d#*, *case2v#* y *op#*. En principio se ha considerado que las operaciones primitivas poseen aridad dos, pero sería necesario añadir una regla por cada aridad de las diferentes operaciones primitivas.

   En la regla *op#*, $k_1\, op\, k_2$ significa el cálculo del valor resultante de la operación primitiva *op* sobre los primitivos k_1 y k_2.

6. Puesto que el lenguaje posee una expresión **let** no recursiva, hay 2 reglas *let* y *letrec* en la Figura 2.8. El comportamiento de la evaluación de la expresión **let** es similar al de la evaluación de la expresión **letrec**, con la salvedad de que el entorno que asocia a las clausuras del *heap* es el original.

2.4. Añadiendo enteros y otros tipos primitivos

La introducción de los enteros y otros tipos primitivos en los lenguajes funcionales perezosos siempre ocasiona problemas. Se suele hablar de dos tipos de valores primitivos: los encapsulados y los desenacapsulados (véase [Lau93, Ses97, PL91]). Normalmente, se suele considerar que todos los valores almacenados en el *heap* son clausuras y, por tanto, han de ser encapsulados. Las ventajas de tener todos los valores encapsulados son que éstos no tienen porque estar definidos y que, por otro lado, el tratamiento se realiza uniformemente. Sin embargo, uno de los principales inconvenientes es que las operaciones primitivas suelen considerar que trabajan sobre enteros desencapsulados. Consecuentemente, es necesario desencapsular dichos enteros y en ocasiones esto lleva a ciertas ineficiencias.

La principal diferencia entre los valores primitivos encapsulados y desencapsulados consiste en que los primeros, al ser clausuras, pueden no estar definidos, es decir, no tener un valor, o este todavía no haber sido calculado. Por su parte, los valores primitivos desencapsulados son directamente un conjunto de bits que se interpreta como un valor. Los valores encapsulados se suelen construir añadiendo un constructor delante del valor desencapsulado, pues de dicha forma se obtiene un valor encapsulado y, por tanto, a la hora de tipar el programa se puede comprobar si es un entero desencapsulado o encapsulado.

Todo esto a su vez, suele llevar a tener varias expresiones **case**: la que trabaja con valores desencapsulados y la que trabaja con valores encapsulados. También suele llevar a tener varias operaciones primitivas: la que trabaja con enteros desencapsulados y la que trabaja con enteros encapsulados. Todas estas complicaciones son transparentes para el programador, ya que desde su punto de vista se puede considerar que todos son enteros y el compilador se encarga de diferenciarlos.

Ejemplo 2.1 Si consideramos los enteros dentro del compilador existirían dos tipos `Int` e `Int#`. Los primeros serían los encapsulados y los segundos los desencapsulados. Las operaciones de suma sobre dichos enteros serían a su vez `+` y `+#`. La `+#(::Int# -> Int# -> Int#)` es la operación primitiva que referencia a la que posee el procesador aritmético del ordenador. Por tanto, la operación `+` se definiría de la siguiente forma:

```
+ :: Int -> Int -> Int
+ x1 x2 = case x1 of
```

```
                Int x1_ -> case x2 of
                             Int x2_ -> Int (x1_ +# x2_)
```

que en STGL se correspondería con la siguiente λ-abstracción:

```
+ :: Int -> Int -> Int
+ = \x1 x2. case x1 of
                Int x1_ -> case x2 of
                             Int x2_ -> case x1_ +# x2_ of
                                          sol_ -> let sol = Int sol_
                                                  in sol
```

□

Esto nos lleva a un amplio abanico de teorías relacionadas con las optimizaciones de dichas operaciones. Para cada función que trabaje con los valores primitivos encapsulados se debería crear, a nivel del compilador, la equivalente que trabaje con desencapsulados; y si ese valor primitivo va a ser utilizado nuevamente para otro cómputo se podría pasar dicho valor primitivo desencapsulado a la función que trabaje con desencapsulados. De esa manera, nos ahorramos la tarea de encapsular y desencapsular dicho valor primitivo.

Pasemos primero a ver en detalle cómo tratan los valores primitivos Launchbury, Sestoft y la máquina abstracta *STG*. Posteriormente comentaremos diversas opciones de optimización sobre ellos al analizar las diferentes observaciones que produce la librería original Hood dependiendo de si se encuentran encapsulados o desencapsulados.

2.4.1. Launchbury y los primitivos

Launchbury introduce los enteros de forma sencilla, ya que no se preocupa de las dos opciones mencionadas anteriormente, encapsulados y desencapsulados. Considera los enteros como constructores de aridad 0. Por esto, no necesita introducir expresiones **case** sobre valores primitivos, ya que son constructores y, por tanto, las expresiones **case** los evalúan perfectamente. Gracias a ello, a nivel sintáctico sólo es necesario añadir las operaciones primitivas, es decir:

$$\begin{array}{rcl} e & \rightarrow & op\ x_1\ x_2 \quad \text{-- operaciones primitivas de aridad 2} \\ & | & \ldots \end{array}$$

A nivel semántico es necesario dar semántica a las operaciones sobre dichos valores primitivos. La nueva regla semántica que trabaja sobre las operaciones primitivas es la siguiente:

$$\frac{H : x_1 \Downarrow K : m_1 \quad K : x_2 \Downarrow L : m_2}{H : op\ x_1\ x_2 \Downarrow L : m_1\ op\ m_2} \quad Prim$$

En esta regla $m_1\ op\ m_2$ significa el valor resultante de aplicar la operación primitiva sobre los dos valores primitivos m_1 y m_2.

Como se puede observar, al no preocuparse sobre el encapsulamiento y desencapsulamiento de dichos valores, Launchbury genera reglas muy claras y sencillas de comprender.

Posteriormente, en el artículo [PL91], publicado junto con Peyton Jones, realiza un estudio de los valores primitivos encapsulados y desencapsulados. En este caso demuestra la equivalencia entre la semántica de dichos primitivos y la semántica denotacional. En dicho artículo se presentan muchas de las ideas y optimizaciones que veremos posteriormente, junto con un sistema de tipos que tiene en cuenta el encapsulamiento y desencapsulamiento de dichos valores.

2.4.2. Sestoft y los primitivos

El artículo [Ses97] de Sestoft es posterior al artículo [PL91]. En dicho artículo Sestoft ya menciona la aproximación tomada por Launchbury y Peyton Jones sobre los valores primitivos encapsulados y desencapsulados.

Sestoft modifica su lenguaje considerando los primitivos como valores desencapsulados, por lo que tiene que añadir a su lenguaje dichos valores y las operaciones sobre dichos valores primitivos, es decir:

$$\begin{array}{rcll} e & \rightarrow & m & \text{-- valores primitivos} \\ & | & op\# \ e_1 \ e_2 & \text{-- operaciones primitivas} \\ & | & \ldots & \end{array}$$

En este caso, Sestoft no normaliza las expresiones asociadas a las operaciones primitivas, ya que no van a ser compartidas. A nivel semántico, al considerar los enteros como valores desencapsulados, Sestoft necesita añadir una regla que indique que son *whnf*. Las nuevas reglas semánticas que trabajan sobre las operaciones primitivas son las siguientes:

$$H : m \Downarrow H : m \qquad Int$$

$$\frac{H : e_1 \Downarrow K : m_1 \quad K : e_2 \Downarrow L : m_2}{H : op\# \ e_1 \ e_2 \Downarrow L : m_1 \ op\# \ m_2} \quad Prim$$

En esta regla $m_1 \ op\# \ m_2$ significa el valor resultante de aplicar la operación primitiva sobre los dos valores primitivos m_1 y m_2.

Por otro lado, Sestoft utiliza los constructores para representar los valores primitivos encapsulados, mientras que utiliza las expresiones **case** para desencapsularlos y evaluarlos antes de aplicar la operación primitiva sobre ellos. Por tanto, un valor encapsulado tiene que ser evaluado para obtener su valor. Dicho valor puede que no exista si este no alcanza una forma normal. Ahora bien, un valor desencapsulado es directamente un primitivo. Sestoft representa los enteros encapsulados con un constructor que lo encapsula, *Int* m. *Int* se corresponde con un constructor de aridad 1 y m representa el valor desencapsulado. Por tanto, a nivel de sus máquinas abstractas su operación + que trabaja sobre enteros encapsulados queda codificada de la siguiente forma:

```
+ =\x1.\x2. case x1 of
              Int n1 -> case x2 of
                          Int n2 -> Int (n1 +# n2)
```

Como se puede ver, Sestoft no introduce nuevas expresiones **case** que trabajen sobre los valores primitivos. A nivel práctico esto elimina muchas de las posibles aplicaciones de dichos valores, ya que no es posible discriminar según el valor del entero.

Para comprender mejor las diferencias en Sestoft entre los enteros encapsulados y desencapsulados veamos un par de ejemplos. En el primero mostraremos la evaluación de una operación con enteros desencapsulados y en el segundo una operación con enteros encapsulados:

Ejemplo 2.2 En este ejemplo mostraremos el cómputo de una operación simple sobre enteros desencapsulados:

$$\dfrac{\dfrac{}{\{\} : 1\# \Downarrow \{\} : 1\#}\ Int \quad \dfrac{\dfrac{}{\{\} : 2\# \Downarrow \{\} : 2\#}\ Int \quad \dfrac{}{\{\} : 3\# \Downarrow \{\} : 3\#}\ Int}{\{\} : (+\# \ 2\# \ 3\#) \Downarrow \{\} : 5\#}\ Prim}{\{\} : +\# \ 1\# \ (+\# \ 2\# \ 3\#) \Downarrow \{\} : 6\#}\ Prim$$

□

Ejemplo 2.3 Ahora mostraremos el cómputo de una suma sobre enteros encapsulados en Sestoft. Veremos de forma independiente la reducción de las dos expresiones $+\ p_1$ y p_2:

$$\dfrac{\dfrac{}{\{\ldots\} : \lambda x_1 \ldots \Downarrow \{\ldots\} : \lambda x_1 \ldots}\ Lam \quad \dfrac{}{\{\ldots\} : \lambda x_2 \ldots [p_1/x_1] \Downarrow \{\ldots\} : \lambda x_2 \ldots [p_1/x_1]}\ Lam}{\{\ldots\} : +\ p_1 \Downarrow \{\ldots\} : \lambda x_2 \ldots [p_1/x_1]}\ App$$

$$\dfrac{\dfrac{\dfrac{}{\{\ldots\} : p_1 \Downarrow \{\ldots\} : Int\ 1}\ Var \quad \dfrac{\dfrac{}{\{\ldots\} : p_2 \Downarrow \{\ldots\} : Int\ 5\#}\ Var \quad \dfrac{}{\{\ldots\} : Int\ (1\#\ +\#\ 5\#) \Downarrow \{\ldots\} : Int\ 6\#}\ Cons}{\{\ldots\} : \textbf{case}\ p_2\ \textbf{of}\ \ldots \Downarrow \{\ldots\} : Int\ 6\#}\ Case}{\{\ldots\} : \textbf{case}\ p_1\ \textbf{of}\ \ldots \Downarrow \{\ldots\} : Int\ 6\#}\ Case$$

Finalmente el cómputo que se produce es el siguiente:

$$\dfrac{\dfrac{}{\{\ldots\} : +\ p_1 \Downarrow \{\ldots\} : \lambda x_2 \ldots [p_1/x_1]}\ App \quad \dfrac{}{\{\ldots\} : \textbf{case}\ p_2\ \textbf{of}\ \ldots \Downarrow \{\ldots\} : Int\ 6\#}\ Case}{\begin{Bmatrix} p_1 \mapsto Int\ 1\# \\ p_2 \mapsto Int\ 5\# \end{Bmatrix} : (+\ p_1)\ p_2 \Downarrow \begin{Bmatrix} p_1 \mapsto Int\ 1\# \\ p_2 \mapsto Int\ 5\# \end{Bmatrix} : Int\ 6\#}\ App$$

□

2.4.3. STG y los primitivos

Como se ha visto anteriormente, la máquina STG trata directamente con los primitivos porque esta máquina pretende modelar el comportamiento del compilador GHC y, por tanto, tiene en cuenta detalles de más bajo nivel.

Los enteros que almacena en las clausuras se corresponden con enteros encapsulados, mientras que los enteros desencapsulados se corresponden con los valores etiquetados con la marca #. Por tanto, la máquina STG considera que para cada operación de bajo nivel existe una operación que trata con los correspondientes valores encapsulados, desencapsulándolos, llamando a la operación primitiva y posteriormente encapsulándolos.

Por este motivo la máquina *STG* tiene que duplicar muchas reglas.

Ejemplo de evaluación de una suma sobre enteros desencapsulados

Para comprender mejor la evaluación de los valores primitivos y sus operaciones en la máquina *STG*, creemos conveniente mostrar la evaluación de una operación primitiva sobre dos valores enteros encapsulados.

Ejemplo 2.4 A continuación pasaremos a presentar la expresión inicial de partida, primero en lenguaje Haskell y posteriormente la traducida al lenguaje STGL. En este ejemplo pretendemos observar la operación suma + aplicada a dos enteros encapsulados. Para ello mostraremos en el ejemplo el código de la operación suma que trabaja sobre enteros desencapsulados y consideraremos que es una clausura más que se almacena en el *heap*. El código Haskell sería simplemente el siguiente:

```
1 + 5
```

que en el lenguaje STGL tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
   + = \x1 x2. case x1 of
                Int x1# -> case x2 of
                           Int x2# -> case x1# +# x2# of
                                      sol# -> let sol = Int sol#
                                              in sol
   uno   = Int 1
   cinco = Int 5
in + uno cinco
```

Los nombres de las variables se han elegido apropiadamente con la intención de hacer el código más comprensible. Obviamente, un generador automático de nombres crearía unos nombres de variables más ilegibles.

Pasaremos ahora a ver la evaluación de dicha expresión en la máquina *STG*. Para hacerla más comprensible, sólo mostraremos el *heap* cuando este varíe y en algunas ocasiones utilizaremos los puntos suspensivos en las expresiones para no escribirlas completas.

	Heap	Control	Entorno	Pila	regla
	$\{\}$	e_0	$\{\}$	$[\,]$	*letrec*
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (\mathit{Int}\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (\mathit{Int}\ x,\ \{x \mapsto 5\#\}) \end{Bmatrix}$	$+\ \mathit{uno}\ \mathit{cinco}$	$\begin{Bmatrix} + \mapsto p_0 \\ \mathit{cinco} \mapsto p_2 \\ \mathit{unoO} \mapsto p_3 \end{Bmatrix}$	$[\,]$	*app1*
$\Longrightarrow$	$\{\ldots\}$	$+$	$\begin{Bmatrix} + \mapsto p_0 \\ \mathit{cinco} \mapsto p_2 \\ \mathit{unoO} \mapsto p_3 \end{Bmatrix}$	$[p_1, p_2]$	*app2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x_1\ \textbf{of}\ \mathit{alts}$	$\begin{Bmatrix} x_1 \mapsto p_1 \\ x_2 \mapsto p_2 \end{Bmatrix}$	$[\,]$	*case1*
$\Longrightarrow$	$\{\ldots\}$	x_1	$\begin{Bmatrix} x_1 \mapsto p_1 \\ x_2 \mapsto p_2 \end{Bmatrix}$	$[(\mathit{alts},\ \{x_2 \mapsto p_2\})]$	*case2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x_2\ \textbf{of}\ \mathit{alts}'$	$\begin{Bmatrix} x_2 \mapsto p_2 \\ x_1\# \mapsto 1\# \end{Bmatrix}$	$[\,]$	*case1*
$\Longrightarrow$	$\{\ldots\}$	x_2	$\begin{Bmatrix} x_2 \mapsto p_2 \\ x_1\# \mapsto 1\# \end{Bmatrix}$	$[(\mathit{alts}',\ \{x_1\# \mapsto 1\#\})]$	*case2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x1\#\ +\#\ x2\#\ \textbf{of}\ \mathit{alts}''$	$\begin{Bmatrix} x_1\# \mapsto 1\# \\ x_2\# \mapsto 5\# \end{Bmatrix}$	$[\,]$	*case1*
$\Longrightarrow$	$\{\ldots\}$	$x1\#\ +\#\ x2\#$	$\begin{Bmatrix} x_1\# \mapsto 1\# \\ x_2\# \mapsto 5\# \end{Bmatrix}$	$[(\mathit{alts}'',\ \{\})]$	*op#*
$\Longrightarrow$	$\{\ldots\}$	$6\#$	$\begin{Bmatrix} x_1\# \mapsto 1\# \\ x_2\# \mapsto 5\# \end{Bmatrix}$	$[(\mathit{alts}'',\ \{\})]$	*case2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{letrec}\ \mathit{sol} = \ \ldots$	$\{\mathit{sol}\# \mapsto 6\#\}$	$[\,]$	*letrec*
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ x_1\ \ldots,\ \{\}) \\ p_1 \mapsto (\mathit{Int}\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (\mathit{Int}\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (\mathit{Int}\ x,\ \{x \mapsto 6\#\}) \end{Bmatrix}$	sol	$\{\mathit{sol} \mapsto p_3\}$	$[\,]$	

Como puede observarse son necesarias once reglas aplicadas de la máquina para reducir una operación primitiva que trabaje sobre enteros encapsulados. La sucesión de pasos viene dada por los siguientes cómputos:

1. Se colocan los argumentos en la pila, como si fuera una aplicación estándar (*letrec*, *app1*, *app2*).
2. Se demanda la evaluación del primer argumento de la operación a forma normal (*case1*).
3. Cuando este alcanza la forma normal se almacena su valor desencapsulado en el entorno (*case2*).
4. Se demanda la evaluación del segundo argumento de la operación a forma normal (*case1*).
5. Cuando este alcanza la forma normal se almacena su valor desencapsulado en el entorno (*case2*).
6. Se realiza la operación primitiva sobre los dos valores desencapsulados (*case1*, *op#*).
7. Se almacena el resultado en el entorno (*case2*).
8. Se crea una clausura para encapsular el valor y almacenarlo en el *heap* (*letrec*).

□

2.4.4. Optimizaciones referentes a los primitivos

En esta sección pretendemos dar una idea de las posibles optimizaciones que se pueden aplicar sobre los valores primitivos. No pretendemos con ello abarcar todas las posibles optimizaciones, sino clarificar la diferencia entre ambos tipos de datos, encapsulados y desencapsulados. Para ello partiremos de varias funciones Haskell y veremos las posibles optimizaciones que sobre ellas se pueden aplicar. Para más información véase el artículo [PL91].

Veamos alguna de las funciones clásicas de Haskell que se utilizarán en los ejemplos y que se pueden optimizar de forma sencilla dependiendo del tipo de valores que reciben (encapsulados o desencapsulados).

Ejemplo 2.5 La función `length` es una función clásica de Haskell que nos devuelve la longitud de una lista, es decir, un entero. Su declaración en Haskell es la siguiente:

```
length :: [a] -> Int
length (x:xs) = 1 + length xs
length []     = 0
```

Esta función devuelve un entero encapsulado. En ese código, tanto la función `+` como los enteros `0` y `1` hacen referencia a la función suma vista anteriormente y a los enteros encapsulados `Int 0` e `Int 1`. Esta se correspondería con la forma uniforme de tratar dichos *valores primitivos*.

En este caso existen varias optimizaciones posibles. La más sencilla podría ser la siguiente:

```
length :: [a] -> Int
length (x:xs) = case length xs of
                  Int sol# -> Int (1# +# sol#)
length []     = Int 0#
```

que tiene la ventaja que de no tener que encapsular el entero 1 en cada llamada recursiva para su posterior desencapsulamiento. Ésta transformación no sería visible para el programador.

Sin embargo, sería más eficiente realizar la siguiente optimización: crear una función que realizará el cálculo y otra función que se encargará de empaquetar el resultado:

```
length :: [a] -> Int
length xs = Int (lenght# xs)

length# :: [a] -> Int#
length# (x:xs) = 1# +# (length# xs)
length# []     = 0#
```

de esta manera se obtienen dos funciones, la original, función `length` original, que devuelve un entero encapsulado y la función `length#` optimizada, que devuelve un entero desencapsulado. No sólo se evita encapsular y desencapsular el entero 1, sino que también se evita el encapsular los resultados alcanzados en cada cómputo recursivo. Además, el compilador a partir de este momento puede analizar el código y averiguar si la llamada a dicha función conviene mantenerla desencapsulada o de lo contrario, conviene encapsularla, según donde vaya a ser utilizada.

En el siguiente caso conviene mantenerlo encapsulado ya que su valor se almacenará en una clausura que posteriormente puede ser demandada desde muchos puntos:

```
letrec
    y = length (1:2:3:4:[])
    ...
in ...
```

En el siguiente ejemplo, sin embargo, conviene mantenerlo desencapsulado y no encapsularlo nunca ya que dicho entero se corresponde con un cómputo local y sólo se utiliza para continuar con la evaluación:

```
case length (1:2:3:4:[]) of
    1 -> ...
    2 -> ...
    otherwise -> ...
```

Por tanto, en este último caso el compilador interpreta que el programador ha escrito lo siguiente:

```
case length# (1:2:3:4:[]) of
    1# -> ...
    2# -> ...
    otherwise -> ...
```

En ambos ejemplos la lista `(1:2:3:4:[])` puede considerarse como una lista con cualquiera de los tipos `[Int]` o `[Int#]`. Sin embargo, Haskell considera la lista como una lista del tipo `[Int]`, ya que Haskell no hace optimizaciones particulares sobre los tipos polimórficos. Es más, en el compilador GHC no existen las listas de valores desencapsulados. El usuario se puede crear su propia lista de enteros desencapsulados definiendo apropiadamente el tipo de datos, que sería el siguiente:

```
data ListIntU =   Cons !Int ListIntU
                | Nil
```

El operador `!`, de bajo nivel, significa que el entero es un entero desencapsulado. A partir de este tipo se podría crear la función length que trabajara con dicha lista, que lleva almacenados enteros desencapsulados. Dicha función sería la siguiente:

```
lengthI :: ListIntU -> Int
lengthI xs = Int (lenghtI# xs)
```

```
lengthI# :: ListIntU -> Int#
lengthI# (Const x xs) = 1# +# (lengthI# xs)
lengthI# Nil          = 0#
```

En algunas situaciones, se puede considerar que es más ventajoso este tipo de listas que el original de Haskell, ya que evita crear clausuras para cada uno de los elementos de la lista, mientras que en el primero hemos de crear dichas clausuras. Por otro lado, en los ejemplos expuestos, dicha lista no va a ser compartida con otras partes del código, por tanto, carece de interés mantener dichas clausuras en el *heap*. Por estos motivos, a veces el segundo tipo de datos sería más eficiente que el primero.

Sin embargo, en muchos casos estas optimizaciones por parte del programador no serían posibles. Incluso, en alguno de los casos que parecen óptimas, podrían resultar perjudiciales para distintas optimizaciones que se producen en el código que produce dicho compilador; es más, al ser una lista estricta este tipo de datos no puede almacenar cálculos perezosos cuya evaluación no concluiría. □

Ejemplo 2.6 Veamos otra de las funciones que se utilizarán mucho en los ejemplos de los siguientes capítulos. La función `sum` que suma los elementos de una lista de números. Su declaración en Haskell es la siguiente:

```
sum :: (Num a) => [a] -> a
sum (x:xs) = x + sum xs
sum []     = 0
```

Esta función devuelve un número encapsulado. En ese código, tanto la función `+` como el entero `0` hacen referencia a la función suma vista anteriormente y al entero encapsulado `Int 0`. Esta se correspondería con la forma uniforme de tratar dichos *valores primitivos*. Sin embargo, sería más eficiente realizar una optimización similar a la vista con la función `length`. La optimización, al igual que en el caso anterior, sería la siguiente: crear una función que realizará el cálculo y otra función que se encargará de empaquetar el resultado:

```
sumInt :: [Int] -> Int
sumInt xs = Int (sumInt# xs)

sumInt# :: [Int] -> Int#
sumInt# (x:xs) = case x of
                    Int x# -> x# +# (xumInt# xs)
sumInt# []     = 0#
```

En este caso, sería necesario una especialización de la función original a cada tipo de datos básicos. De esta forma, no es necesario encapsular los resultados parciales. Por tanto, se evita la encapsulación de los resultados alcanzados en cada caso recursivo, es decir, `length lista -1` elementos. Esta optimización es algo más complicada, pero un compilador podría realizarla sin problemas. Es más, se podría generar también la función que se presenta a continuación, que no sólo devuelve un entero desencapsulado, sino que trabaja con una lista de enteros desencapsulados.:

```
sumInt# :: [Int#] -> Int#
sumInt# (x:xs) = x +# (sumInt# xs)
sumInt# []     = 0#
```

□

Las optimizaciones vistas en este último ejemplo no las realiza el compilador GHC, ya que, como se comentó previamente, este compilador no realiza ninguna optimización sobre los tipos polimórficos, porque es una tarea complicada y a veces no se obtienen las mejoras deseadas. Sin embargo, la optimización vista en el ejemplo previo sí la realiza el compilador GHC. De hecho, el compilador suele hacer un análisis de estrictez sobre las funciones para averiguar qué argumentos son estrictos y realizar ciertas optimizaciones sobre ellos. En el caso de que dichos argumentos sean valores primitivos, el compilador genera, al igual que el ejemplo anterior, una función que se encarga de empaquetar el resultado y una función que realiza el cálculo. Además, como se comentó previamente, el compilador analiza en cada momento si puede mantener el valor desencapsulado o por el contrario es necesario encapsularlo para mandárselo a la siguiente función.

Como puede verse, el estudio de la inclusión de los valores primitivos conlleva un estudio exhaustivo sobre las posibles optimizaciones que sobre ellos se pueden aplicar. Dicho estudio, es decir, el estudio de las multiples posibilidades de optimización según si el primitivo se encuentra encapsulado o desencapsulado, queda fuera de las pretensiones de esta tesis. Eso no significa que no incluiremos los valores primitivos, ya que su inclusión nos permitirá:

1. Por un lado, desarrollar los ejemplos de forma más clara y comprensible.

2. Por otro lado, acercarnos un poco más a la compilación real y no dejar aislados los valores primitivos, que a nuestro modo de ver son importantes y sin embargo no suelen aparecer a nivel semántico, aunque sí a nivel de la implementación.

Como los consideramos muy importantes, a nivel del estudio de la depuración en la máquina *STG*, en el Capítulo 5, mantendremos tanto los valores primitivos encapsuladoscomo los desencapsulados. De esta forma se podrán aplicar las optimizaciones vistas anteriormente en dicha máquina.

Sin embargo, debido a que trabajar con valores encapsulados y desencapsulados conlleva la inclusión de nuevas expresiones **case** que trabajen sobre ellos y a la duplicación de varias reglas, y que esta duplicación, desde nuestro punto de vista, sólo aleja la atención de los detalles más relevantes de la inclusión de la depuración, nosotros incluiremos a nivel semántico los valores primitivos de una forma más uniforme. Así, mantendremos mayor claridad en los ejemplos y no enturbiaremos las reglas semánticas con detalles excesivos. Seguiremos, en este caso, las ideas de Launchbury, que, como hemos visto, considera los valores primitivos como constructores "especiales" de aridad `0`. Para que, por un lado, las expresiones **case** trabajen sobre ellos y no sea necesario duplicarlas y, por otro lado, no sera necesario el trabajo de encapsulamiento y desencapsulamiento que a nivel semántico no es relevante.

Por otro lado, aunque no hagamos un estudio semántico en detalle del encapsulamiento y desencapsulamiento, analizaremos las observaciones que produce la librería Hood sobre dichas funciones que trabajan con valores primitivos desencapusulados. Ya que la aplicación de dicha librería sobre dichos valores dará como resultado observaciones cuanto menos llamativas. Para más detalles véase el Capítulo 3.4.2.

Capítulo 3

Una introducción a los depuradores funcionales

En un mundo ideal en el que los programas se diseñan correctamente desde el principio, no existen errores de transcripción, por lo que la depuración de programas no sería necesaria. Lamentablemente no es el caso y, por tanto, se puede considerar que el proceso de depuración es inherente al proceso de programación. La depuración en programación imperativa puede desarrollarse de muy diversas formas:

- Mediante instrucciones de escritura durante la ejecución del programa, para que este imprima ciertas marcas que indican por dónde pasa la ejecución del programa y mostrando la información de las variables necesarias.
- Mediante un entorno de depuración que muestre los valores de las variables, que permita introducir puntos de ruptura, evolución paso a paso, etc.
- Mediante la visualización del árbol de llamadas.
- Mediante la visualización de los datos mostrando cómo se generan o destruyen.

En los lenguajes funcionales impacientes la depuración también consiste en una tarea relativamente simple y similar al caso imperativo, ya que la impaciencia hace que los argumentos se evalúen antes de evaluar la función. Sin embargo, la depuración en programación funcional perezosa resulta mucho más difícil. La tarea de la depuración en este área suele ser compleja pues, entre otras cosas, la evaluación perezosa suele llevar a cabo los cálculos en un orden que puede ser difícil de comprender. Por otro lado, normalmente cualquier intento por observar los datos puede conllevar la modificación del comportamiento perezoso de dichos lenguajes. Es por ese motivo entre otros por lo que inicialmente no se prestó mucha atención al desarrollo de depuradores para los lenguajes funcionales perezosos.

Una de las primeras aproximaciones a este problema generó los depuradores basados en la observación del grafo del cómputo. Entre ellos, podemos destacar ART (Advanced Redex Trails) que después se convirtió en Hat [WCBR01, CRW02, Run07]. Sin embargo, tanto la dificultad para realizar dicha depuración como la de interpretar los resultados hicieron que esta primera aproximación se desechara.

Uno de los primeros depuradores que aparecieron en programación funcional perezosa fueron los depuradores declarativos. En estos no se veían los pasos de evaluación que se realizaban en el cómputo, sino las reducciones semánticas que se habían producido; es decir, los resultados de la aplicación de una función a sus argumentos. Entre estos depuradores destacaremos Freja [NS97, Nil01, Nil07] y Buddha [Pop98, Pop04, Pop07]. Ambos trataban de averiguar el código incorrecto a través de las respuestas que obtenían del programador a las preguntas que generaban de forma automática.

Posteriormente aparecieron otros depuradores, entre los que destacamos Hood[Gil00a, Gil00b] y Hat [WCBR01, CRW02, Run07]. Hood surgió con la idea de poder observar el cómputo perezoso sin modificar el comportamiento perezoso de los programas. Se creó como una librería independiente de Haskell y lo único necesario para utilizarlo era cargar la librería y anotar las expresiones a observar. Se puede considerar como una instrucción de escritura perezosa, que observa el cómputo de las estructuras marcadas. Hat no sólo modificó su nombre sino que fue uno de los depuradores que más ha cambiado y se ha mantenido a lo largo de los años. Por un lado, incorporó las ideas de otros depuradores y aportó nuevas herramientas para la depuración. La idea principal de esta herramienta consistía en analizar el grafo generado por el cómputo tras la evaluación del programa. Una comparación detallada de los depuradores expuestos anteriormente se encuentra en [CRW01].

Recientemente se han diseñado nuevos depuradores basados en la misma idea de depuración que existe en los lenguajes imperativos: puntos de ruptura y análisis de los valores de las variables en dichos puntos. Este tipo de depuración es algo más compleja que en programación imperativa ya que la evaluación de las "variables" puede que no haya llegado a su fin y, por tanto, los resultados que se observan son parciales. El programador que utilice este tipo de depuradores tiene que conocer en mayor profundidad cómo se evalúa en programación perezosa y qué resultados cabe esperar que se observen.

En este capítulo presentaremos un breve resumen de los depuradores mencionados anteriormente para que el lector interesado se haga una idea general de la tarea de depuración en los lenguajes funcionales perezosos. Nuestra pretension no es mostrar exhaustivamente todos los depuradores, ni todos los detalles de dichos depuradores. Presentaremos con mayor grado de detalle el depurador Hood, ya que este se considera uno de los pilares básicos de esta tesis. Por tanto, primero veremos las ideas principales del resto de depuradores presentados anteriormente y finalmente nos centraremos en el depurador Hood junto con su extensión gráfica GHood [Rei01, Rei07].

3.1. Depuradores declarativos

Una de las aproximaciones que ha surgido con bastante fuerza en los lenguajes de alto nivel ha sido la idea de generar un depurador semi-automático que con poca ayuda del usuario encuentre el error en el código de la aplicación. Gracias al mayor nivel de abstracción de los lenguajes declarativos es más sencillo que en los lenguajes imperativos averiguar dónde se ha producido el error.

¿Qué significa que un depurador sea declarativo? En términos generales un depurador declarativo es aquel depurador que se basa en la semántica declarativa del programa en vez de en los pasos que se realizan en su cómputo. En lenguajes de alto nivel (funcionales, lógicos, lógico-funcionales, etc) es muy importante basarse en la semántica, ya que los pasos de evalua-

ción necesarios para realizar los cómputos suelen ser difíciles de comprender incluso para un programador experto. Por ese motivo, en este tipo de lenguajes han proliferado este tipo de depuradores. La idea de la depuración declarativa proviene de la comunidad de programación lógica. A modo de ejemplo, en [Sha83, Tes96, Cab05, MSB05] el lector interesado puede encontrar tanto los orígenes de la declaración declarativa, como diversas aproximaciones a la depuración declarativa tanto en lenguajes lógicos como en lenguajes lógico-funcionales.

Concentrándonos en el campo de la programación funcional perezosa, prestaremos especial atención a dos depuradores declarativos: Freja y Buddha. Ambos son depuradores declarativos para el lenguaje funcional Haskell. Están basados en preguntas del tipo sí/no, donde las respuestas del usuario hacen que se acote el fragmento de código a analizar, de tal forma que mediante estas respuestas el programa devuelve la función que está incorrectamente implementada. Para ello se supone que el usuario (es decir, el programador) debe conocer la especificación de todas las funciones del programa. Las preguntas realizadas por los depuradores corresponden a las reducciones de las funciones, es decir, el depurador pregunta si el resultado de una función es el esperado con respecto a los datos de entrada a los que se aplicó en ejecución.

El método utilizado por ambos depuradores es similar y a grandes rasgos consiste en lo siguiente:

- Realizar una transformación inicial en el código del programa.
- Compilar el programa transformado.
- Unirlo a una librería de depuración.
- Ejecutar el programa. Durante esta ejecución se crea el árbol de reducciones semánticas.
- Comenzar la sesión con el usuario que consiste en que el usuario vaya contestando a las preguntas que el depurador propone automáticamente hasta dar con la función o funciones incorrectas.

Las principales características de ambos depuradores son las siguientes:

Buddha

- Ejecuta todo el programa antes de comenzar con la sesión de depuración.
- No posee depuración para las funciones que realizan entrada y salida de datos.
- Permite depurar el estándar Haskell 98.
- Funciona en el compilador GHC.

Freja

- Ejecuta el programa paso a paso mientras está en la sesión de depuración.
- Posee depuración para las funciones que realizan entrada y salida de datos. Esto es así porque al ejecutar a la vez que el usuario se encuentra en la sesión de depuración, se hace más sencillo la depuración de estas funciones.
- Sólo funciona bajo el sistema operativo Solaris.
- Sólo permite la depuración de un subconjunto de Haskell 98.

El principal problema de estos depuradores es que la depuración de un programa grande es complicada, ya que la cantidad de reducciones que se han producido es grande y, por tanto, también lo es la cantidad de respuestas demandadas al usuario. Además necesitan mucho espacio de memoria para almacenar la información de las reducciones que han sucedido.

3.2. El depurador Hat

En esta sección mostraremos las ideas básicas acerca del depurador Hat. Hat surgió inicialmente como una herramienta para detectar errores en los programas y comprender mejor su comportamiento. Para dicho propósito mostraba el grafo de reducciones. Pero este mecanismo resultaba bastante complicado de entender. Desde su nacimiento ha resultado ser la herramienta de depuración de Haskell que más ha evolucionado, incorporando para ello las ideas de otros depuradores.

Actualmente Hat se puede considerar como un conjunto de herramientas. Para ejecutar este depurador, al igual que en Freja y Buddha, es necesario compilar el programa de forma especial para que cuando se ejecute genere una traza en un fichero. El comportamiento inicial del programa se mantiene salvo que genera dicho fichero como efecto lateral. Cabe resaltar que, aunque el programa finalice con un error o sea abortado por el usuario, se genera la traza. Por tanto, puede ser analizada con las herramientas para ver el motivo de error o la situación en la que se encontraba el cómputo.

Las principales ventajas de este depurador consisten en que no es necesario modificar el código para realizar la depuración y en que posee un amplio abanico de posibilidades para analizar el comportamiento del programa. Cada herramienta nos presenta la traza desde distintos puntos de vista y posee 5 herramientas que veremos en detalle. Sin embargo, su principal desventaja es que el archivo que genera es demasiado grande, a diferencia de Hood que genera ficheros pequeños concentrados sólo en la parte de traza que desea ver el usuario.

A continuación pasaremos a describir el comportamiento de las herramientas de Hat.

Hat-observe. Consiste en una herramienta basada en el depurador Hood. A diferencia de Hood, en esta herramienta sólo podemos observar las aplicaciones de las funciones que se encuentran definidas en el nivel principal del módulo.

Las observaciones que se obtienen con esta herramienta consisten en la representación de todas las aplicaciones de la función seleccionada, es decir, a diferencia de Hood, no se puede observar ni las aplicaciones concretas que se producen en un cómputo, ni las estructuras de datos intermedias.

Hat-trail. Es en una herramienta basada en el análisis del grafo generado en el fichero. Esta herramienta permite sólo el análisis interactivo desde el resultado a las reducciones que nos llevaron a obtener dicho resultado.

En este caso el usuario va dirigiendo la visualización de la parte del grafo que le interesa para descubrir el error en su código.

Hat-explore. Al igual que Hat-trail, consiste en una herramienta basada en el análisis del grafo generado en el fichero. La diferencia principal con respecto a Hat-trail es que esta herramienta permite la exploración del grafo en ambas direcciones, es decir, desde cada función permite observar a qué funciones ha llamado y cuál ha sido la función que la ha llamado. Además resalta el código causante de dicha evaluación. Por tanto, nos va dando información paulatina de qué código es responsable de dicho resultado.

Se trata, por tanto, de la herramienta más cercana a un depurador convencional imperativo, con la ventaja de que permite viajar en ambas direcciones. Además, permite anotar las

expresiones indicando si son correctas o incorrectas. De esta manera se puede ir acotando el código generador del error.

Hat-detect. Esta herramienta está inspirada en Freja.

Al igual que Freja y Buddha, se trata de un depurador declarativo en el que partiendo del resultado del programa y de forma semi-automática detecta el lugar donde se encuentra el error. El usuario sólo tiene que contestar a preguntas del tipo sí/no, es decir, pregunta si una función bajo un cierto argumento debería dar como resultado el valor que dicha función nos devolvió. Si la respuesta es incorrecta se pregunta al usuario sobre las repuestas que devolvieron las funciones a las que dicha función llamó. Por tanto, va desde el resultado hacia los cómputos internos.

Hat-stack. Esta herramienta permite examinar la pila de llamadas para cómputos que han terminado erróneamente, es decir, parados por el usuario o con un mensaje de error.

Esta herramienta parte del error y muestra la pila de las últimas llamadas que se han producido. De esta forma podemos hacernos una idea de los motivos por los que se ha producido el error. Para entender mejor las llamadas, la pila de llamadas que muestra no es la que corresponde con el cómputo perezoso, sino la que se hubiera producido en caso de que estuviéramos realizando el cómputo en un entorno impaciente.

3.3. Depuradores pseudo-imperativos

Recientemente ha surgido una nueva área de investigación en el entorno de los depuradores de los lenguajes funcionales perezosos. La idea principal de este área de investigación consiste en conseguir depuradores cuyo manejo sea similar a los depuradores para los lenguajes imperativos, es decir, parar, examinar y continuar el cómputo; estableciendo los puntos de ruptura adecuados, etc. En este área podemos destacar los depuradores HsDebug [EP03b], Rectus [MK06] y un depurador integrado en el interprete GHCi [Him06, IM07].

HsDebug. HsDebug está implementado bajo la técnica de depuración de programas perezosos siendo impaciente, es decir, realiza parte de los cómputos de forma impaciente, para que la pila que se genere corresponda con la pila que se hubiera producido en impaciente y, por tanto, sea fácilmente comprensible. Esta herramienta añade impaciencia en un programa perezoso de forma segura, realizando una cantidad finita de pasos. Así, es sencillo mantener la corrección de los programas perezosos incluso aunque tengan estructuras infinitas.

Este depurador está basado en deshabilitar la eliminación de las llamadas recursivas de la pila y la modificación del RTS del compilador GHC, en vez de modificar el código del programa. De esta manera las transformaciones del programa y optimizaciones se pueden aplicar sin problemas. Funciona sobre cualquier programa Haskell y permite añadir puntos de ruptura simples, puntos de ruptura con una guarda, examinar el estado, realizar llamadas a funciones para ver su resultado, mostrar el estado de la pila, realizar paso a paso la ejecución y continuar con la ejecución normal. Este depurador permite la depuración de las mónadas y de las funciones que realizan entrada y salida de datos.

Su principal problema radica en que la modificación del RTS resulta una tarea muy costosa y difícil de mantener, teniendo en cuenta que el compilador GHC está constantemente evolucionando.

Rectus. Rectus está basado en las mismas ideas que HsDebug: evaluar impacientemente de forma segura los programas para poder depurar al estilo imperativo. Sin embargo, este depurador funciona como herramienta independiente del compilador GHC y está implementado realizando una transformación del programa que hace impaciente el código, en vez de evaluar de forma impaciente las expresiones a reducir.

Permite las mismas opciones de depuración que HsDebug y facilita la depuración de mónadas y funciones que realizan entrada y salida de datos. Actualmente existe un prototipo independiente del compilador GHC.

Su principal problema es que las transformaciones necesarias en el código original del programa son complejas y su mecanismo de evaluación conlleva una alta sobrecarga.

Depuradores integrados en GHCi. Una de las últimas ideas que han surgido en el área de la depuración consiste en integrar la herramienta de depuración dentro del intérprete del compilador GHC (GHCi). De esta manera se hace más sencillo su uso, ya que no es necesario una herramienta extra o cargar una librería encargada de la depuración. Además, se consigue que el mantenimiento de la herramienta se realice en paralelo a las distintas versiones del compilador.

Para realizar esta tarea de forma sencilla se parte de la premisa de que no se quiere modificar ni el RTS, ni la evaluación perezosa. Esto conlleva ciertas limitaciones y dificultades a la hora de realizar la depuración.

La herramienta de depuración que se está integrando dentro de GHCi permite establecer puntos de ruptura, con o sin guarda; examinar estructuras intermedias para averiguar si se encuentran suspendidas debido a la pereza; forzar la evaluación de la clausura; cambiar el valor de las variables mutables; ejecutar paso a paso; o continuar con la evaluación.

El principal problema de este depurador es que, debido a que mantiene el orden de evaluación perezoso, resulta complicado crear y manejar adecuadamente la pila que se produciría en un entorno impaciente. Por ahora, ésta es un área abierta sin una solución adoptada.

Como puede verse, este área de investigación es reciente y cada una de las aproximaciones vistas en esta sección poseen diferentes problemas. Por otro lado, estos depuradores no cubren adecuadamente todas las opciones de depuración vistas en los demás depuradores.

3.4. El depurador Hood

En esta sección mostraremos las ideas básicas acerca de la librería Hood. Para más detalles sobre dicha librería el lector interesado puede consultar [Gil00a, Gil00b].

Cuando estamos depurando programas escritos en un lenguaje imperativo, el programador no sólo puede explorar el resultado final del cómputo, sino que también puede analizar los valores intermedios almacenados en las variables de programa. Además, es relativamente simple seguir las modificaciones de cada variable a lo largo del tiempo.

Desafortunadamente, esta tarea no es tan simple cuando estamos trabajando con lenguajes funcionales perezosos. Sin embargo, Hood es una librería creada con la intención de realizar unas observaciones similares a las que se realizan en imperativo de forma sencilla y clara. De hecho, Hood provee una forma de observar cualquier estructura intermedia que aparezca en un programa. Además, utilizando la herramienta GHood [Rei01], que presentaremos más adelante, también es posible observar la evolución de las estructuras bajo observación a lo largo del tiempo.

Para ilustrar el tipo de observaciones que se obtienen con la librería Hood creemos conveniente presentar el ejemplo expuesto en [Gil00a]. Es suficientemente complejo como para resaltar los aspectos más importantes de Hood, pero también es relativamente simple para entenderlo de forma clara sin necesitar un conocimiento exhaustivo de Haskell.

Dado un número natural, la siguiente función de Haskell retorna la lista de todos los dígitos en base 10 del número:

```
natural :: Int -> [Int]
natural = reverse
          . map (`mod` 10)
          . takeWhile (/= 0)
          . iterate (`div` 10)
```

Es decir, `natural 3408` devuelve la lista `3:4:0:8:[]`, donde `[]` representa la lista vacía y `:` representa el constructor de las listas. Recalcaremos que para obtener el resultado final se generan tres listas intermedias en el siguiente orden:

```
-- after iterate
    3408:340:34:3:0:_
-- after takeWhile
    3408:340:34:3:[]
-- after map
    8:0:4:3:[]
```

Nótese que la primera lista es una lista infinita, aunque sólo se computan los primeros cinco elementos. El resto de la lista no es necesario evaluarlo, por tanto, no es demandado, es decir, no llega a computarse. Por ese motivo se representa con `_` que significa que dicho elemento no se redujo.

Utilizando Hood podemos anotar el programa para obtener las listas intermedias mostradas anteriormente. Para que se muestren dichas listas tenemos que utilizar la función `observe` que es la función central de la librería Hood. La declaración de tipo de dicha función es la siguiente:

```
observe :: (Observable a) => String -> a -> a
```

Desde el punto de vista de la evaluación, `observe` sólo retorna su segundo argumento, es decir, `observe s a = a`. Sin embargo, como efecto lateral, una vez que se evalúa la clausura correspondiente al valor `a`, se almacena su valor resultante junto con la etiqueta `s`. Recalcaremos que la función `observe` devuelve su segundo parámetro de manera perezosa, es decir, que el grado de evaluación de `a` no es modificado por la introducción de la observación, de igual forma que no es modificado cuando se aplica la función identidad `id`. De esta forma, como el árbol de evaluación no se modifica, Hood puede trabajar con listas infinitas, tal y como se presentó anteriormente en la aplicación `iterate (`div` 10)`.

Si consideramos de nuevo el ejemplo simple propuesto anteriormente, con esta nueva función podemos observar todas las estructuras intermedias introduciendo tres anotaciones de observación tal y como se muestra a continuación:

```
natural :: Int -> [Int]
natural = reverse
            . observe "after map"
            . map (`mod` 10)
            . observe "after takeWhile"
            . takeWhile (/= 0)
            . observe "after iterate"
            . iterate (`div` 10)
```

Después de la ejecución de `natural 3408`, obtendremos el resultado deseado. Hood no sólo nos muestra este tipo simple de estructuras vistas anteriormente. De hecho, puede observar cualquier otra estructura que aparezca en un programa Haskell. En particular, con esta librería se pueden observar funciones. Por ejemplo,

```
observe "sum" sum (4:2:5:[])
```

observará la aplicación de la función `sum` a sus parámetros, mostrando la siguiente observación:

```
-- sum
  { \ (4:2:5:[]) -> 11
  }
```

Nótese que la observación anterior puede leerse como que *cuando la función recibe como argumento de entrada la lista `4:2:5:[]` devuelve como salida el valor `11`*. Los elementos de la lista `4`, `2` y `5` aparecen de forma explícita ya que fueron demandados para obtener la salida. Sin embargo, si observamos lo siguiente:

```
observe "length" length (4:2:5:[])
```

obtendremos la siguiente observación:

```
-- length
   { \ (_:_:_:[]) -> 3
   }
```

Es decir, estamos observando una función que cuando recibe una lista con tres elementos devuelve el número 3. Para dicho cálculo no ha necesitado evaluar el valor concreto de cada elemento de la lista, únicamente ha necesitado evaluar la estructura de la lista para averiguar su número de elementos.

Tal y como se puede esperar, las funciones de orden superior también pueden ser observadas. Esto se realiza de la misma forma que en los casos anteriores. Por ejemplo, en nuestro ejemplo inicial, en vez de observar las estructuras intermedias podemos observar la función de orden superior `iterate`:[1]

```
natural :: Int -> [Int]
natural = reverse
            . map (`mod` 10)
            . takeWhile (/= 0)
            . observe "iterate" iterate (`div` 10)
```

En esta situación, cuando aplicamos la función `natural` sobre el número `3408`, Hood devuelve:

[1] Esta función de orden superior aplica infinitas veces la primera función que recibe. Por ejemplo, si aplicamos `iterate (+3) 1` devuelve la siguiente lista infinita `1:4:7:10:13:...`

```
-- iterate
   { \ { \ 3 -> 0
       , \ 34 -> 3
       , \ 340 -> 34
       , \ 3408 -> 340
       } 3408
        -> 3408 : 340 : 34 : 3 : 0 : _
   }
```

Es decir, se observa que es una función que devuelve `3408:340:34:3:0:_` cuando recibe como segundo parámetro `3408` y como primer parámetro la función `‘div‘ 10` que ha sido observada para los cuatro valores de entrada recibidos: `3408`, `340`, `34` y `3`.

Finalmente resaltaremos un detalle de la implementación de Hood. Aunque está garantizado que la función `observe` no modifica el árbol de evaluación de los valores observados, Hood puede cambiar el comportamiento de los programas en cuestión de espacio, ya que pierde la compartición de sus réplicas, es decir, cuando la evaluación de una clausura bajo observación alcanza la forma normal correspondiente, se crea una nueva clausura cuyo valor coincide con el de la anterior forma normal, modificada adecuadamente, indicando que se encuentra bajo observación. Desde ese momento se pierde la compartición de dicha clausura con el resto del programa. Obviamente esto es ineficiente, principalmente en términos de memoria, pero también en términos de tiempo, ya que es necesario crear una nueva clausura. Sin embargo, debido a esta pérdida de compartición, se hace más sencillo decidir quién es el responsable de la evaluación de cada componente. Por tanto, si estamos observando una estructura en un entorno dado, no estamos interesados en las partes de dicha estructura que han sido evaluadas desde otros entornos. Por ejemplo, si estamos observando la función `length` en el siguiente ejemplo

```
let xs = take 5 (1:2:3:4:5:6:7:[])
in (observe "length" length xs) + (sum xs)
```

obtendremos la siguiente salida:

```
-- length
   { \ (_:_:_:_:_:[]) -> 5
   }
```

Es decir, incluso aunque todos los elementos de la lista `xs` han sido computados por la función `sum`, no han sido necesarios para el cómputo de la función `length` y, por tanto, no han sido observados.

3.4.1. Hugs-Hood

El intérprete de Haskell Hugs posee una versión modificada de Hood cuyo comportamiento es bastante diferente cuando se está observando una expresión anotada desde varios entornos. De hecho, si se observa la función `length` en la misma situación presentada anteriormente, la observación que muestra corresponde con la siguiente:

```
-- length
   { \ (1:2:3:4:5:[]) -> 5
   }
```

Es decir, Hugs no sólo observa lo que ha sido realmente demandado por la función `length`, sino que también observa lo que ha sido evaluado de la expresión anotada con la observación. La

principal ventaja de utilizar las observaciones de Hugs consiste en que en la implementación de Hugs de la librería Hood se mantiene la compartición de las clausuras. Además, se obtiene una información diferente: el grado de evaluación de cada una de las estructuras en el programa. Sin embargo, se pierde la información sobre qué función es la responsable de la evaluación de esos datos, que en la librería original de Hood se mantiene.

Resumiendo, tanto la versión original de Hood como la versión implementada en Hugs poseen ciertas ventajas sobre la otra. Por tanto, consideramos relevante proveer de una semántica formal para ambas aproximaciones e incluso tratar de obtener las mejoras de ambas. Esto se verá en detalle en el Capítulo 6.

3.4.2. Hood y los primitivos

En la Sección 2.4 se vieron en detalle los valores primitivos y las posibles optimizaciones que sobre ellos se pueden realizar. Creemos conveniente presentar aquí ciertos detalles no documentados de la librería Hood cuando es utilizada sobre estos valores. Para ello utilizaremos una de las funciones vista en dicha sección: la función `lentghI#`. Recordaremos que esta función trabajaba sobre una lista de enteros desencapsulados y nos devolvía su longitud.

La observación de la función `lentghI#` utilizando la librería Hood daría resultados curiosos. Aunque dicha función no demanda la evaluación de los elementos de la lista, estos se observarían, a diferencia de si estos hubieran estado encapsulados, ya que no son clausuras, son valores básicos ya evaluados. Por tanto, si evaluamos una expresión como:

```
observe "lengthI#"  lengthI# (Cons 1# (Cons 2# (Cons 3# Nil)))
```

obtendríamos la siguiente observación:

```
-- lengthI#
   \ (1:2:3:[]) -> 3
```

Esto puede entenderse como un pequeño error de dicha librería, que al estar implementada como una librería externa al compilador no puede diferenciar claramente entre un valor desencapsulado y una clausura. Por ese motivo, al observar un valor desencapsulado éste directamente se observa. En nuestra semántica procuraremos evitar este tipo de problemas, ya que consideramos que la filosofía subyacente a Hood consiste en sólo observar lo que realmente demanda una función.

3.4.3. Detalles de la implementación de Hood

Para comprender mejor el comportamiento de Hood es necesario analizar el tipo de observaciones que esta librería genera. Los detalles expuestos en esta sección surgen del análisis del código de dicha librería.

Las anotaciones que provoca Hood cuando el programa está siendo observado siguen el siguiente patrón: (*portId*, *parent*, *change*). El *portId* corresponde con un puntero al lugar donde se encuentra la anotación: en la implementación corresponde con un valor entero. Este puntero apunta a la anotación previa a la que esta haga referencia y es nulo (0 en la implementación) si no hace referencia a ninguna anotación previa. Para poder realizar un postproceso de las observaciones, cuando una función o un constructor se evalúa se produce una anotación que referencia a una anotación previa y cada uno de sus argumentos se etiqueta indicando el lugar donde se ha producido dicha anotación junto con el número de argumento correspondiente. Esta es la

información que se almacena en *parent* (es decir, quién es el padre de la anotación); formalmente, *parent* es una tupla (*observeParent*, *observePort*), donde *observeParent* corresponde con el puntero *portId* del padre y *observePort* corresponde con la posición del argumento. Finalmente, el parámetro *change* corresponde con el tipo de observación que se ha llevado a cabo, que posee una de las siguientes formas:

- *Observe String* se genera la primera vez que se entra en una clausura anotada con dicha marca de observación. Este tipo de observación es el único que no posee padre, por tanto hace referencia al padre nulo, que en la implementación corresponde con el valor (0, 0). Esta es la primera anotación que se produce cuando se está observando una clausura y comienza su reducción.

- *Enter* se genera cuando la evaluación de una clausura comienza.

- *Cons Int String* se genera cuando la clausura se reduce a un constructor. El entero que aparece en la anotación representa la aridad del constructor, mientras que el String representa el nombre del constructor. Los hijos del constructor son anotados con (*parentPortId*, 1), (*parentPortId*, 2), ... (*parentPortId*, *arity*). De esta manera, es sencillo reconstruir el árbol de evaluación donde *parentPortId* corresponde con el puntero al lugar donde se acaba de producir la anotación *Cons*.

- *Fun* se genera cuando la evaluación de la clausura se reduce a una función. En las observaciones que se producen en Hood las λ-abstracciones poseen únicamente un argumento y un único valor resultante. Cuando la evaluación finaliza, se analiza el árbol para obtener la lista de argumentos y resultados. El argumento de la λ-abstracción se anota con el siguiente padre; (*parentPortId*, 0) y el resultado de la λ-abstracción se anota con (*parentPortId*, 1) donde *parentPortId* hace referencia al lugar donde se ha producido la anotación *Fun*.

Por tanto, en Hood no sólo es posible observar las formas normales de las clausuras sino que también se observa cuándo se comienza la evaluación de las clausuras. Utilizando esta información es sencillo averiguar qué clausura es demandada por otra. Esta es la forma en la que trabaja GHood, que muestra las clausuras bajo evaluación.

En Hood las anotaciones mostradas anteriormente son procesadas y presentadas de forma clara al usuario tal y como se vio en la Sección 3.4, es decir, de forma plana y textual. Esto se consigue gracias a la función `runO :: IO a -> IO ()` implementada en dicha librería, que se encarga de pasar al modo de depuración, inicializando adecuadamente las variables necesarias para que las observaciones se almacenen y procesando las observaciones obtenidas a lo largo del cómputo. Por tanto, para obtener las observaciones es necesario aplicar dicha función sobre el programa.

3.5. GHood

GHood [Rei01] es sólo una herramienta gráfica implementada en Java que dado un fichero con las anotaciones presentadas en la versión anterior genera una gráfica en la que se puede ver la evolución paso a paso de las anotaciones, tanto hacia adelante como hacia atrás. La modificación necesaria en la librería de observaciones Hood consiste únicamente en que, una vez obtenidas las

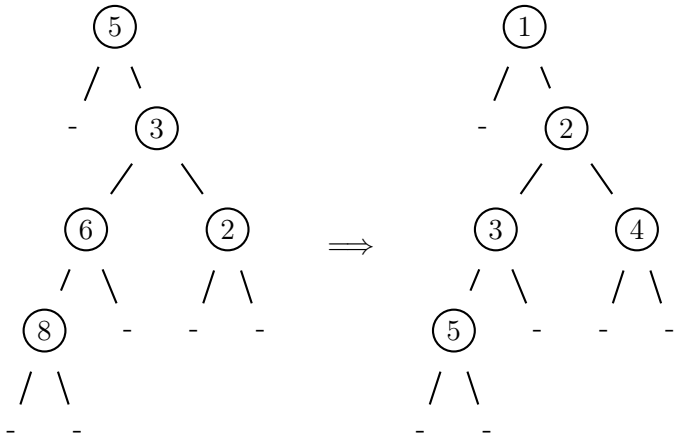


Figura 3.1: Ejemplo de numeración de árbol binario

anotaciones, se genera un fichero donde dichas anotaciones se vuelcan, y a continuación se abre la herramienta que carga dicho fichero y genera una visualización de dichas observaciones.

Para comprender mejor qué tipo de visualización gráfica genera GHood, partiremos del ejemplo presentado en [Rei01]. El problema consiste en numerar ordenadamente por niveles los nodos de un árbol binario. En la Figura 3.1 podemos observar a la izquierda el árbol binario original y a la derecha el árbol resultante tras la numeración.

Supongamos que la función `numera` se encarga de numerar dicho árbol y nosotros estamos interesados en ver el comportamiento de dicha función, es decir, si primero recorre todo el árbol y luego va reconstruyendo el árbol resultante, o si va creando el árbol resultante a la vez por niveles, etc. Para ello sería necesario incluir una observación antes y después de la función `numera` y analizar paso a paso las observaciones que obtenemos, es decir, aplicando las siguientes modificaciones `observe "despues" $ numera $ observe "antes" arbol`. En Hood en ambos casos obtendríamos la siguiente anotación plana:

```
-- antes
     N V _ (N (N (N V _ V) _ V) _ (N V _ V))
-- despues
     N V _ (N (N (N V 5 V) 3 V) 2 (N V 4 V))
```

donde `N` es el constructor del árbol binario y `V` corresponde con una hoja del árbol. Como se puede ver, para devolver su resultado, la función `numera` no ha necesitado evaluar los valores almacenados en las hojas del árbol. Pero no sabemos si primero ha recorrido todo el árbol y posteriormente ha comenzado a generar el nuevo. GHood nos permite visualizar este tipo de cosas. En este caso la herramienta nos permite viajar a través de las 56 observaciones que ha producido la evaluación. En la Figura 3.2 se presentan las transiciones 14, 15 y 16 de dichas observaciones. En rojo con la palabra *thunk* se muestran las clausuras que no han sido demandadas. El color amarillo sobre una clausura de tipo *thunk* indica que se encuentra demandada y, por tanto, bajo evaluación. A la izquierda nos encontramos los botones relativos a la visualización y encima nos encontramos un par de parámetros de visualización configurables: la escala y el retardo entre tiempos. En la primera imagen podemos observar que se encuentran dos nodos de los hijos bajo

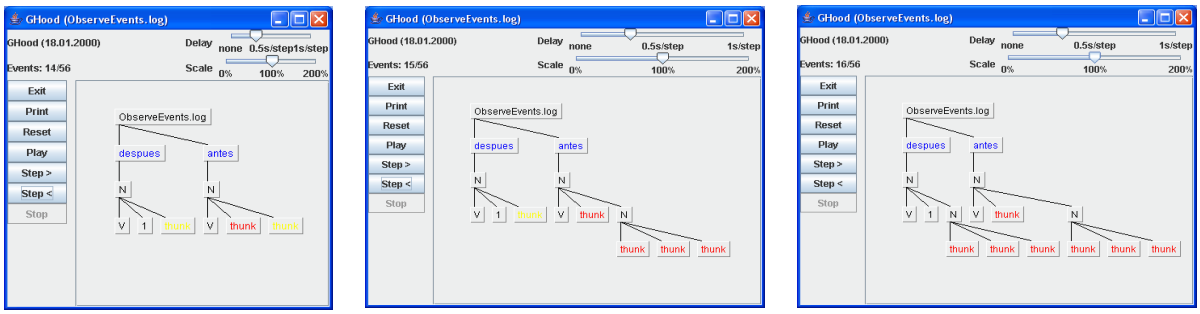


Figura 3.2: GHood

evaluación. En verdad el nodo del árbol de "despues" demandó la evaluación del árbol de "antes", como puede apreciarse en la secuencia. En la segunda imagen podemos apreciar que el árbol de "antes" se ha reducido parcialmente en el nuevo nodo y, por tanto, deja de estar demandado y muestra su valor. En la tercera figura podemos observar que el árbol "despues" genera un nuevo nodo y deja de estar bajo evaluación, ya que en este punto ya se sabe que hay un nuevo nivel. Por tanto, podemos concluir que esta versión de la función `numera` va creando el árbol por niveles a la vez que va analizando el árbol viejo.

Como puede apreciarse, con esta herramienta se puede ver una información que no se observa con Hood. La visión temporal de los acontecimientos nos da información relativa a qué clausuras demandan otras clausuras, y en el caso de las funciones nos indica qué tipo de recorridos realizan sobre los datos o qué tipo de recursión realizan, etc.

Esta herramienta podrá ser utilizada sin ninguna modificación en la implementación de la extensión paralela que veremos en el Capítulo 7.

Capítulo 4

Extensiones paralelas de Haskell

Cabe destacar que uno de los problemas menos tratados hasta ahora en el ámbito de la programación funcional paralela es la depuración. De hecho, en la actualidad no existe ningún depurador en programación funcional perezosa paralela. Es por este motivo que en esta tesis desarrollaremos el primer depurador en programación funcional.

La forma de abordar la solución en paralelo a un problema consiste en especificar cómo deben repartirse los distintos cómputos entre los distintos procesos, en vez de indicar únicamente qué valor debe computar el programa. La ejecución de un programa paralelo lleva consigo muchos aspectos, tales como la creación de hebras, la ejecución en cada procesador, la transferencia de datos a un proceso y desde un proceso, la sincronización entre procesos, etc. El manejo de todos estos aspectos para crear un algoritmo paralelo correcto y eficiente hace muy costosa (en tiempo de programación) la aproximación explícita al paralelismo. La otra aproximación diametralmente opuesta consiste en únicamente manejar a nivel del sistema de control de ejecución (runtime system, a partir de ahora RTS) todos estos aspectos, sin necesidad de que el programador se ocupe de ellos. Desafortunadamente, esta aproximación implícita no suele ser muy eficiente en tiempo de ejecución. Así pues, es difícil crear programas que manejen de forma explícita el paralelismo y si lo realizamos de forma implícita entonces los programas no suelen ser eficientes. De hecho, se han desarrollado muchos lenguajes de programación funcional paralela, tanto con paralelismo explícito como con paralelismo implícito. Entre medias de ambos tipos de programación paralela existe una gran diversidad de posibilidades según se deje al programador el manejo de parte de los detalles de organización del cómputo.

A lo largo de los años, en programación funcional se han ido desarrollando varias extensiones paralelas de entre las que cabe mencionar: Ph (parallel Haskell) [NA01], Caliban [Kel87], Eden [BLOP98], GpH [THJ+96], Nepal [CKLP01], Data Parallel Haskell [Hil94], Data Field Haskell [HL00], HPF [KLS94] y HDC [Her00]. Cada una de estas aproximaciones deja en manos del programador distintos detalles de la organización del cómputo. La realización de una comparativa entre estos lenguajes queda fuera del alcance de esta tesis. Ahora bien, el lector interesado puede consultar [TLP02], donde se presenta una comparativa muy detallada sobre un gran número de estos lenguajes funcionales paralelos.

En el ámbito de esta tesis nos centraremos en los lenguajes GpH y Eden, que son aproximaciones intermedias entre la aproximación explícita e implícita al paralelismo. Además, poseen propiedades muy interesantes. Ambos son extensiones paralelas de Haskell que con poco trabajo del programador dan como resultado considerables tasas de aceleración. Además, ambos se

compilan a partir del compilador GHC [Pey96]. GpH está basado en que el programador anote las expresiones que pueden calcularse en paralelo, mientras que Eden está basado en la creación de procesos por el programador. Así, GpH está más próximo a los lenguajes de paralelismo implícito, mientras que Eden es más explícito. Por otro lado, la semántica operacional de ambos lenguajes es una extensión de la semántica operacional estándar de Haskell, y refleja la distinción entre los sublenguajes de cómputo y de coordinación que conviven en él. Por tanto, el trabajo que se realizará para añadir la depuración en GpH se aprovechará para incorporar la depuración en Eden, tanto a nivel de implementación como a nivel semántico.

En este capítulo presentaremos las características más relevantes de ambas extensiones paralelas que son necesarias para el desarrollo de la extensión paralela del depurador y para los ejemplos de depuración que veremos en el Capítulo 7. En los capítulos posteriores implementaremos y daremos semántica a la depuración basada en observaciones sobre dichos lenguajes paralelos. Además, utilizaremos las observaciones para analizar la especulación en el lenguaje Eden.

4.1. GpH

GpH (Glasgow Parallel Haskell [THJ+96, THLP98]) es una modesta extensión conservativa del lenguaje Haskell basada en hebras, que añade un nuevo combinador de composición paralela `` `par` ``, que al ser utilizado junto a la composición secuencial `` `seq` `` permite controlar el grado de paralelismo de los programas. Las principales características del lenguaje son las siguientes:

- Los procesos se comunican mediante el *heap*.
- La comunicación es implícita (el paso de mensajes se realiza de modo automático, no necesita que el programador la especifique mediante instrucciones del estilo *send* y *receive*), asíncrona (se envían los datos sin necesidad de esperar la aceptación del receptor) y de tipo *pull* (primero se demanda un dato y luego se obtiene).
- Permite la introducción de cómputos especulativos.
- No pueden realizarse cómputos no-deterministas.
- No permite el desarrollo de programas reactivos.
- A nivel semántico se asume la existencia de una memoria compartida. A nivel de implementación, el *heap* se encuentra distribuido entre los procesadores, pero garantizando que cada parte del *heap* está controlada por uno único.
- Se garantiza que la implementación no duplica trabajo (si bien el programa puede forzar a que un trabajo se duplique).

Este tipo de aproximación al paralelismo permite crear hebras, pero no provee mecanismos para controlar dichas hebras. El manejo de las hebras lo realiza el RTS. La operación `` `par` `` es considerada la operación generadora del paralelismo y se utiliza en muchas variantes de lenguajes paralelos. El RTS, sin embargo, puede ignorar cualquier tipo de paralelismo. En este tipo de modelos el programador sólo indica qué expresiones del programa pueden ser evaluadas de forma

eficiente en paralelo. Recalcaremos que el RTS maneja los detalles de la ejecución paralela, tales como la creación de las hebras, la comunicación, etc. Combinando dichas primitivas simples junto con las funciones de orden superior, se pueden construir abstracciones de alto nivel, tales como las estrategias de evaluación propuestas en [THLP98]. Dichas estrategias no se utilizarán en esta tesis, ya que a nivel semántico sólo se consideran azúcar sintáctico.

La semántica de los combinadores `‘par‘:: a -> b -> b` y `‘seq‘ :: a -> b -> b` es la siguiente:

$$\begin{array}{lllll} \bot & \text{`seq`} & y & = & \bot \\ x & \text{`seq`} & y & = & y \\ x & \text{`par`} & y & = & y \end{array}$$

Denotacionalmente, los operadores `‘par‘` y `‘seq‘` son proyecciones en su segundo argumento. Operacionalmente, `‘seq‘` fuerza la evaluación a *whnf* de su primer argumento antes de evaluar el segundo argumento, mientras que `‘par‘` indica que la evaluación a *whnf* de su primer argumento puede realizarse en paralelo con la evaluación del segundo argumento, aunque será en tiempo de ejecución cuando se decida si se crea o no una nueva hebra para evaluar el primer argumento de `‘par‘`. Normalmente, el primer argumento será necesario para la evaluación del segundo, pues de lo contrario se estaría realizando una evaluación completamente especulativa. Nótese que el uso de `‘par‘` es completamente transparente desde el punto de vista semántico, es decir, los resultados obtenidos serán los mismos independientemente de que se introduzcan o no combinadores `‘par‘`.

Buena parte de la eficiencia de GpH se debe a que está implementado sobre el eficiente compilador GHC. De hecho, la implementación de GpH sólo modifica el RTS de GHC para introducir las estructuras necesarias que permitan gestionar la distribución paralela del cómputo, conservándose por completo todas las fases de compilación de GHC.

La sincronización entre las distintas hebras viene determinada únicamente por las dependencias de datos: si una hebra necesita un dato que está produciendo otra hebra, entonces se bloquea hasta que haya sido producido (nótese que GpH no duplica trabajo).

Al igual que en Eden, para establecer las comunicaciones entre los distintos procesadores, se puede elegir usar la conocida librería de paso de mensajes PVM [GBD+94] o la librería MPI [GLS99], lo cual facilita notablemente la portabilidad de la implementación. Además, para mejorar la eficiencia, los datos no se envían uno a uno, sino que se empaquetan por bloques, con el objetivo de reducir el impacto negativo que pueden provocar altas latencias entre los procesadores.

4.1.1. Ejemplo escrito en GpH

A modo de ejemplo, consideremos el siguiente programa para calcular números de Fibonacci:

```
parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 ‘par‘ (nf1 ‘seq‘ (nf1+nf2))
           where nf1 = parfib (n-1)
                 nf2 = parfib (n-2)
```

Las definiciones de los casos base son evidentes, pero el caso general merece una explicación más detallada. Debe leerse como: puede crearse una nueva hebra para evaluar `nf2`, pero `nf1` se evaluará en el procesador actual antes de realizar la suma final. Si se creó una nueva hebra para `nf2`, entonces será la responsable de evaluarlo y de comunicar el resultado a la hebra principal.

Si no se creó la hebra, entonces todo el trabajo lo realizará la hebra principal. La sincronización se realiza cuando se procesa el cómputo relativo a la suma: si el cómputo de `nf1` ha terminado antes que el de `nf2`, entonces la hebra encargada de hacer la suma debe bloquearse en espera de que la otra hebra termine de evaluar `nf2`.

Realmente, el uso de las anotaciones `` `par` `` y `` `seq` `` proporciona una gran flexibilidad a la hora de realizar programas paralelos, pero tiene como contrapartida el hecho de que la labor de programación resulta demasiado laboriosa, y el comportamiento paralelo de los programas es difícilmente comprensible. Afortunadamente, el uso de *estrategias* [THLP98] facilita enormemente la programación en GpH.

4.2. Eden

Eden [BLOP96, BLOP97, BLOP98, PR01, LOP05] es también una extensión del lenguaje funcional perezoso Haskell que incorpora construcciones para definir e instanciar procesos de forma *explícita*. De esta manera se consiguen desarrollar programas tanto concurrentes como paralelos. Las principales características del lenguaje son las siguientes:

- Los procesos se comunican intercambiando valores a través de canales de comunicación modelados (fundamentalmente) mediante listas perezosas.
- La comunicación es implícita y asíncrona de tipo *push*.
- Permite la introducción de cómputos especulativos.
- Permite el desarrollo de programas reactivos.
- Puede realizar cómputos no-deterministas.
- No asume la existencia de un *heap* compartido, sino que está diseñado para trabajar eficientemente sobre arquitecturas distribuidas.
- Permite duplicar trabajo debido a que los procesos poseen *heaps* disjuntos (independientemente de que se encuentren o no en la misma memoria física).

Este tipo de aproximación al paralelismo hace que Eden sea considerado un lenguaje paralelo parcialmente abstracto de alto nivel, es decir, el programador es el responsable de fijar el grafo de procesos y el reparto de tareas entre los mismos, mientras que las comunicaciones, las sincronizaciones y el manejo de los procesos se realizan de forma implícita (via *streams*, es decir, listas perezosas estrictas en sus elementos).

Mientras que la mayoría de los lenguajes funcionales concurrentes presuponen la existencia de memoria compartida, Eden está diseñado para trabajar eficientemente sobre arquitecturas distribuidas. Además, Eden admite no-determinismo y creación dinámica de canales, lo cual permite modelar redes dinámicas de procesos y sistemas reactivos. No obstante, ambos pueden considerarse como una extensión al lenguaje Eden y no se ven afectados por la depuración, por tanto, no se mostrarán en esta tesis.

Un proceso que toma como entradas $in_1, \ldots, in_m$ y produce como salidas $exp_1, \ldots, exp_n$, se especifica mediante una expresión de *abstracción de proceso* de la siguiente forma:

```
process (\(in_1,...,in_m) -> (exp_1,...,exp_n))
    where ecuacion_1
          ...
          ecuacion_r
```

cuyo tipo es `Process (tau_1,...,tau_m) (tau'_1,...,tau'_n)` donde `Process` es un constructor de tipos binario predefinido, y `(tau_1,...,tau_m)` y `(tau'_1,...,tau'_n)` son, respectivamente, los tipos de las entradas y de las salidas. La parte `where` es opcional, y se utiliza para definir funciones auxiliares y subexpresiones comunes que se utilizan en la definición del proceso.

Realmente, no es necesario que se nombren explícitamente todos y cada uno de los canales de entrada y todas y cada una de las expresiones de salida, sino que para especificar los canales de entrada puede utilizarse cualquier ajuste de patrones (incluyendo una única variable), mientras que la salida del proceso puede ser cualquier expresión. La única restricción impuesta es que los tipos sean correctos:

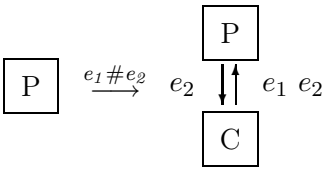
```
process (\entradas -> expresion)
    where ecuaciones
```

Un proceso puede tener como entrada (respectivamente, salida) un único canal o una tupla de canales. Por tanto, para inferir tipos correctamente es preciso aplicar ciertos convenios:

- Si el tipo inferido para `a` (respectivamente, para `b`) en `Process a b` es una tupla, entonces cada uno de los componentes de dicha tupla se considera un canal independiente.
- Si un canal tiene tipo `[a]` (es decir, es una lista), entonces se tratará como una lista (potencialmente infinita) que transmite su contenido elemento a elemento.

La función `process :: (a -> b) -> Process a b` encapsula una función del tipo `(a -> b)` dentro de una *abstracción de proceso* del tipo `Process a b` que cuando se aplica se ejecutará en paralelo. Por otro lado, la instanciación de procesos se realiza mediante el operador infijo `( # ) :: Process a b -> a -> b`. Al igual que en GpH, se pueden definir estrategias para controlar el paralelismo de forma más sencilla. En el caso de Eden es posible definir *esqueletos* [KLPR01]: por ejemplo, pueden desarrollarse diversas implementaciones paralelas del esqueleto `map` que denotacionalmente se comportan igual que la función `map`.

Cuando se invoca una *instanciación de proceso* (`e1 # e2`) se crea un nuevo *entorno de cómputo*. El nuevo proceso (el hijo o el proceso instanciado) es alimentado por su creador enviando el valor de su argumento e_2 via el correspondiente canal de entrada y el proceso hijo devuelve el valor de la aplicación $e_1 e_2$ (a su padre) a través del canal de salida, como muestra el siguiente diagrama:



Destacaremos que las abstracciones de procesos y la instanciación # son valores de primera clase, es decir, pueden usarse igual que cualquier otro valor: pueden utilizarse como parámetros de una función, pueden almacenarse en estructuras de datos, pueden ser el resultado devuelto por una función, etc. Como por ejemplo en

```
zipWith (#) ps ins
```

A nivel semántico las abstracciones de procesos pueden compararse con funciones, la principal diferencia es que cuando éstas se aplican se ejecutan en paralelo. No obstante, desde el punto de vista semántico no hay ninguna diferencia entre las abstracciones de procesos y las λ-abstracciones, la única diferencia radica en su aplicación. Por este motivo, en el lenguaje que presentaremos en el Capítulo 8 utilizaremos λ-abstracciones para definir los procesos.

A continuación resaltaremos algunos detalles respecto a los procesos que consideramos relevantes en Eden:

Impaciencia v.s. pereza.

Aunque el modelo computacional de Eden es un lenguaje perezoso (Haskell), se ha optado por no mantener dicha pereza para el modelo de coordinación, con el objetivo de mejorar el comportamiento paralelo de los programas. Así, la pereza se rompe en dos situaciones:

- Una vez lanzado un proceso, su cómputo está dirigido por la evaluación de sus expresiones de salida, para las cuales siempre existe demanda. De esta forma, se incrementa el grado de paralelismo, puesto que los procesos pueden producir datos de forma independiente, sin necesidad de esperar a que nadie los demande.
- Se permite que un proceso se lance antes de ser demandado, con el objetivo de acelerar la distribución del cómputo.

Nótese que, al romper la evaluación perezosa, las dos reglas anteriores pueden conducir a que se realice trabajo innecesario para el cómputo final, pero es un riesgo que se acepta por la ventaja de mejorar el grado de paralelismo de los programas, si bien el programador deberá tenerlo siempre en mente cuando utilize el lenguaje.

Reparto de trabajo.

Con el objetivo de poder razonar acerca del comportamiento paralelo de los programas, debe quedar claro qué partes del cómputo deben realizarse en cada proceso. En Eden, al evaluarse

```
e1 # e2
```

el padre se encargará de evaluar `e2`, y de comunicar su valor a `e1`, mientras que todo el cómputo necesario para evaluar `e1` será llevado a cabo por el nuevo proceso hijo. Esto incluye no sólo la aplicación de `e1` a los valores de entrada correspondientes a `e2`, sino también la evaluación de las variables libres de `e1` (en caso de que sea preciso evaluarlas). A modo de ejemplo, en

```
p :: Int -> Int -> Process Int Int
p x y = process (\i -> x + y + i)
result = p (fib 5) (fib 6) # (fib 7)
```

el padre sólo se encargará de evaluar `fib 7`, mientras que será el proceso hijo quien evalúe no sólo la suma final, sino también `fib 5` y `fib 6`.

Evaluación de las salidas de los procesos.

El objetivo de un proceso es evaluar sus salidas. Excepto las listas que se transmiten en forma de *streams*, las expresiones se evaluarán *completamente* (es decir, a forma normal) antes de ser enviadas por un canal de salida.

Para cada una de las expresiones de salida de un proceso, se creará un flujo de ejecución concurrente distinto, puesto que, en principio, dichas evaluaciones son independientes. Por tanto,

se distinguen dos niveles de concurrencia en Eden: la concurrencia y evaluación paralela de los procesos; y la evaluación concurrente de las distintas hebras de cada proceso.

Sincronización.

Cuando una hebra concurrente de un proceso necesita un cierto valor de un canal de entrada, pero dicho valor aún no ha sido recibido, la evaluación de dicha hebra será suspendida hasta que el correspondiente emisor produzca y envíe el dato. Nótese que la comunicación a través de canales se realiza utilizando envío no-bloqueante, pero recepción bloqueante, y que la sincronización entre los procesos se realiza única y exclusivamente mediante el intercambio de información a través de los canales.

Esqueletos en Eden.

Eden es un lenguaje que permite la definición de esqueletos como funciones de orden superior. Nosotros en esta tesis sólo resaltaremos los esqueletos que posteriormente utilizaremos. Esqueletos más complejos se pueden encontrar en [Rub01, LOP+02].

El esqueleto más simple es `map`. Dada una lista de entradas `xs` y una función `f` a ser aplicada a cada una de las entradas, la especificación secuencial en Haskell es la siguiente:

```
map f xs  =  [f x | x <- xs]
```

que puede leerse como que para cada elemento `x` de la lista `xs`, debe aplicarse la función `f` a dicho elemento. Esto puede ser trivialmente paralelizado en Eden. Para que cada proceso realice una única tarea, utilizaremos la siguiente aproximación:

```
map_par f xs =  [pf # x | x <- xs] `using` spine
  where pf = process f
```

La abstracción de proceso `pf` encapsula la función `f`. De esta forma se crea un proceso para cada tarea. La estrategia `spine` (véase [THLP98] para detalles) se utiliza para que de forma impaciente se evalúe la espina de la lista de procesos instanciados. Así, todos los procesos se crean de forma inmediata. Sin ella los procesos se crearían bajo demanda.

Cuando los procesos finalizan, envían el resultado al padre, que en este caso es común para todos. Cuando el padre obtiene su valor finaliza el cómputo de los procesos. Este esqueleto puede ser fácilmente mejorable, pues en vez de crear un proceso por cada tarea se puede crear un número fijo de procesos (ej. un proceso por cada procesador) y distribuir las tareas entre los procesos. De esta manera, cuando el proceso finaliza una tarea envía el resultado y pasa directamente a computar la siguiente tarea. A este esqueleto propuesto se le suele llamar `map_farm` ("granja"), cuya implementación es la siguiente:

```
map_farm np unshuffle shuffle f xs
  = shuffle (map_par (map f) (unshuffle np xs))
```

donde `np` corresponde con el número de procesos a crear, la función `unshuffle` es la encargada de particionar la lista de tareas y la función `shuffle` es la encargada de mezclar los resultados. Dependiendo de las estrategias aplicadas para la partición del trabajo se obtienen varias posibilidades de reparto de trabajo. La única restricción que debe cumplirse es que para toda lista `xs`, `shuffle (unshuffle np xs)) == xs`.

En los ejemplos de esta tesis también utilizaremos otra implementación del esqueleto `map` a la que llamaremos "trabajadores replicados"[KPR00]. Este esqueleto consiste en enviar una nueva tarea a cada proceso según éste haya finalizado su cómputo, en vez de ir enviando todas las tareas independientemente de si el trabajador ha terminado o no. Este esqueleto suele estar

parametrizado por un entero que indica el *buffer* que tiene cada proceso, de tal manera que inicialmente se le envían a cada proceso tantas tareas como pueda almacenar en dicho *buffer*. Recalcaremos que esta implementación del esqueleto es beneficiosa cuando el tiempo necesario para realizar las tareas no es uniforme o los procesadores no tienen la misma potencia de cálculo.

4.2.1. Ejemplo escrito en Eden

A modo de ejemplo, mostraremos el mismo programa que en GpH escrito en el lenguaje Eden, es decir, el cálculo de números de Fibonacci:

```
parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = (parfibP # (n-1)) + (parfibP # (n-2))

parfibP = process parfib
```

Al igual que antes, los casos base son evidentes, pero el caso general merece una explicación más detallada. A diferencia de antes, ahora el proceso donde se esté calculando el fibonacci de n no será el responsable de calcular el fibonacci de $n-1$ y el de $n-2$. Se crearán dos procesos que calcularán independientemente el valor del fibonacci de $n-1$ y de $n-2$ respectivamente. Cuando dichos procesos envíen el resultado al padre, él se hará cargo de calcular la suma de ambos valores para calcular el número fibonacci de n. En este caso el padre sólo es responsable de evaluar y enviar a través del canal respectivo el valor $n-1$ o $n-2$. La sincronización se realiza cuando los procesos devuelven los resultados, ya que en ese momento se desbloquea el cómputo de la hebra correspondiente con la suma.

Aunque en este ejemplo no se utilice toda la potencia expresiva de Eden, puede verse que esta extensión proporciona gran flexibilidad a la hora de realizar programas paralelos. Además, en este caso el programador no *sugiere* los puntos donde se pueden realizar los cómputos en paralelo, sino que crea los procesos para que éstos realicen el cómputo.

Capítulo 5

Incorporando facilidades de depuración en la máquina *STG*

En este capítulo estudiaremos diversas posibilidades para incluir características de depuración a nivel de la máquina abstracta *STG*. Las máquinas que presentaremos están basadas en las vistas en la Sección 2.3. Tanto el lenguaje como la máquina que mostraremos aquí son una extensión de las que aparecieron en dicha sección, donde ahora se introducen características para poder tratar las observaciones. El objetivo principal de este capítulo consiste en realizar una primera aproximación a la depuración a nivel de la máquina *STG*, de tal forma que nos sirva de guía para el desarrollo de una semántica abstracta que incluya la depuración y el posterior desarrollo de una máquina equivalente a dicha semántica.

Es en este capítulo donde nos sentiremos libres para explorar las diferentes posibilidades de observación de las estructuras de la máquina abstracta *STG*. Veremos que a nivel de la máquina abstracta se pueden obtener de forma sencilla varias versiones sin necesidad de tener que realizar muchas modificaciones. Posteriormente, en el capítulo siguiente generaremos todo el desarrollo formal de una de dichas versiones, la que se corresponde más fielmente con el comportamiento de la librería Hood.

Las tres versiones del depurador presentadas en este capítulo están inspiradas en las siguientes herramientas de depuración:

- versión original de Hood.
- versión de Hood implementada en el intérprete Hugs.
- versión de Hood con compartición de clausuras.

Dichas versiones se realizarán sobre la máquina *STG*.

El trabajo expuesto en este capítulo ha dado lugar a la publicación [ELR05].

5.1. Incluyendo observaciones en STGL

En primer lugar debemos decidir el mecanismo con el que se añadirán las características de depuración. Para ello introduciremos observaciones en el lenguaje al estilo de Hood, es decir, marcaremos de forma especial las estructuras que queremos observar. En consecuencia, en nuestro

		-- Expresiones	
e	$\rightarrow$	$x\ \overline{atom_i}^n$	-- aplicación
	$\mid$	$op\ \overline{atom_i}^n$	-- operador primitivo saturado
	$\mid$	$atom$	-- variable o literal
	$\mid$	**let** $\overline{x_i = be_i}^n$ **in** e	-- let
	$\mid$	**letrec** $\overline{x_i = be_i}$ **in** e	-- let recursivo
	$\mid$	**case** e **of** $alts$	-- expresión case
		-- Ligaduras	
be	$\rightarrow$	w	-- formas normales débiles de cabeza
	$\mid$	e	-- expresión
	$\mid$	$x^{@str}$	-- **variable observada**
		-- Formas normales débiles de cabeza ($whnf$)	
w	$\rightarrow$	$C\ \overline{atom_i}$	-- aplicación de constructores
	$\mid$	$\lambda\ \overline{x_i}.e$	-- abstracción lambda
		-- Alternativas	
$alts$	$\rightarrow$	$\overline{C_i\ \overline{x_j}^{k_i} \mapsto e_i}^k.default$	-- alternativas algebraicas
	$\mid$	$\overline{prim_i \mapsto e_i}^k.default$	-- alternativas primitivas
$default$	$\rightarrow$	**otherwise** $\mapsto e$	
	$\mid$	$y \mapsto e$	
	$\mid$	ϕ	
		-- Átomos	
$atom$	$\rightarrow$	$x,\ y,\ p,\ q$	-- variables
	$\mid$	$prim$	-- valores primitivos
		-- Primitivos	
$prim$	$\rightarrow$	$int(1\#,\ 2\#,\ \ldots)$	-- enteros primitivos
	$\mid$	$\ldots$	-- otros
		-- Operaciones sobre primitivos	
op	$\rightarrow$	$+\#$	-- suma
	$\mid$	$\ldots$	-- otras

Figura 5.1: Lenguaje STGL-Observado

caso debemos ser capaces de anotar cualquier estructura. Esto puede ser realizado de forma trivial permitiendo anotar como *observable* cualquier variable. Para ello, añadiremos un nuevo tipo de ligadura en las expresiones **let** o **letrec**. De esta forma, sólo es necesario realizar una pequeña modificación en el lenguaje STGL para incluir una nueva ligadura, como se muestra en la Figura 5.1. Consecuentemente, para cada expresión observada existirán dos ligaduras: una con

la expresión a observar y otra con la observación de dicha expresión. Llamaremos a dicho lenguaje STGL-Observado. Véase que $x^{@str}$ es equivalente a la expresión de Hood `observe str x`.

Hemos decidido obligar a que las observaciones aparezcan sólo como ligaduras de una expresión **let** o **letrec**. De esta manera su tratamiento será más sencillo ya que, como veremos más adelante, las marcas de observación, que serán necesarias para su cómputo, no se extenderán sobre las expresiones. Además, de esta forma mantenemos la potencia expresiva del lenguaje, puesto que el único cambio necesario para observar una expresión consiste en vincular dicha expresión en una ligadura de un **let** o **letrec** y generar una nueva ligadura que observe dicha ligadura.

Este lenguaje será utilizado en cada una de las versiones que iremos codificando sobre la máquina abstracta.

5.2. Tratando las observaciones en la máquina *STG*

Una vez que el lenguaje nos permite incluir observaciones, debemos tratar dichas observaciones en la máquina abstracta. La idea principal consiste en que cada regla puede generar efectos laterales. En nuestro caso dichas observaciones van a ser producidas sobre un fichero organizado por líneas. Por tanto, aparece una nueva componente en la máquina abstracta *STG* que es el fichero. Lo primero que hay que hacer es añadir dicho fichero a las reglas originales de la máquina *STG* que se mantienen intactas para nuestros propósitos. En la Figura 5.2 puede observarse dicha modificación.

Recalcaremos que dicho fichero no sufre ninguna modificación mientras evaluamos las expresiones normales de la máquina STG.

En el artículo [Pey92] aparecen dos reglas semánticas para la realización de la aplicación parcial, regla *var2*. La que presentamos en la Figura 5.2 se corresponde con la que realmente se implementa, ya que no realiza la aplicación parcial, sino que almacena en el *heap* una aplicación parcial que posteriormente habrá que reducir, es decir, actualiza el *heap* con una forma que no es considerada una *whnf*. Otra posibilidad sería la de realizar una actualización con la aplicación parcial correspondiente, es decir, añadir al *heap* la siguiente ligadura $q \mapsto (\lambda\overline{y_i}^n.e, E_1 \cup \{\overline{x_i \mapsto p_i}^k)\}|)$. Veremos que esta aproximación puede resultar más interesante a la hora de observar las λ-abstracciones.

Las reglas que se presentan en esa figura son comunes para todas las versiones que presentaremos en este capítulo. La diferencia entre las distintas versiones radica en cómo tratan las marcas de observación.

5.2.1. Codificando Hood en la máquina abstracta *STG*

El tratamiento de las nuevas expresiones, las relativas a las observaciones, producirá anotaciones en el fichero de observaciones. Las anotaciones de observaciones se añaden secuencialmente en el fichero. Por consiguiente, usaremos la notación $f \circ \langle ann \rangle$ para indicar que hemos añadido la anotación *ann* en una nueva línea al final del fichero f. Las anotaciones de observación seguirán el siguiente patrón: $\langle strs, q \mapsto e \rangle$. Donde *strs* se corresponde al conjunto de marcas de observación que están viendo reducir dicha clausura, mientras que $q \mapsto e$ se corresponde con la clausura observada.

	Heap	Control	Entorno	Pila	Efecto lateral	Regla
	H	**let** $\{\overline{x_i = be_i}\}$ **in** e	E	S	f	$let(^1)$
$\Rightarrow$	$H \cup [\overline{q_i \mapsto (be_i, E\|)}]$	e	E_1	S	f	
	H	**letrec** $\{\overline{x_i = be_i}\}$ **in** e	E	S	f	$letrec(^2)$
$\Rightarrow$	$H \cup [\overline{q_i \mapsto (be_i, E_1\|)}]$	e	E_1	S	f	
	H	**case** e **of** $alts$	E	S	f	$case1$
$\Rightarrow$	H	e	E	$(alts, E\|) : S$	f	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts, E_1) : S$	f	$case2(^3)$
$\Rightarrow$	H	e_k	$E_1 \cup \{\overline{y_{ki} \mapsto p_i}\}$	S	f	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.\textbf{otherwise} -> e, E_1) : S$	f	$case2d(^4)$
$\Rightarrow$	H	e	E_1	S	f	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q\}$	$(alts.y -> e, E_1) : S$	f	$case2v(^4)$
$\Rightarrow$	H	e	$E_1 \cup \{y \mapsto q\}$	S	f	
	H	$x\ \overline{x_i}^n$	$E\{\overline{x_i \mapsto p_i}^n\}$	S	f	$app1$
$\Rightarrow$	H	x	E	$\overline{p_i}^n : S$	f	
	$H[q \mapsto (\lambda\overline{x_i}^n.e, E_1)]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^n : S$	f	$app2$
$\Rightarrow$	H	e	$E_1 \cup \{\overline{x_i \mapsto p_i}^n\}$	S	f	
	$H \cup [p \mapsto (e, E_1)]$	x	$E\{x \mapsto p\}$	S	f	$var1$
$\Rightarrow$	H	e	E_1	$\#p : S$	f	
	$H[p \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto p\}$	$\overline{p_i}^k : \#q : S$	f	$var2(^5)$
$\Rightarrow$	$H \cup [q \mapsto (x\ \overline{x_i}^k, E_5\|)]$	x	E	$\overline{p_i}^k : S$	f	
	$H[q \mapsto (C_k\ \overline{x_i}, E_1)]$	x	$E\{x \mapsto q\}$	$\#p : S$	f	$var3$
$\Rightarrow$	$H \cup [p \mapsto (C_k\ \overline{x_i}, E_1)]$	x	$\{x \mapsto p\}$	S	f	
	H	x	$E\{x \mapsto k\}$	S	f	$prim\#$
$\Rightarrow$	H	k	$\{\}$	S	f	
	H	k	E	$(alts, E_1) : S$	f	$case2\#(^3)$
$\Rightarrow$	H	e_k	E_1	S	f	
	H	k	E	$(alts.\textbf{otherwise} -> e, E_1) : S$	f	$case2d\#(^4)$
$\Rightarrow$	H	e	E_1	S	f	
	H	k	E	$(alts.y -> e, E_1) : S$	f	$case2v\#(^4)$
$\Rightarrow$	H	e	$E_1 \cup \{y \mapsto k\}$	S	f	
	H	$op\ x_1\ x_2$	E	S	f	$op\#(^6)$
$\Rightarrow$	H	$k_1 op\ k_2$	$\{\}$	S	f	

(^{1}) $\overline{q_i}$ son distintas y frescas con respecto a H, **let** $\{\overline{x_i = be_i}\}$ **in** e y S. $E_2 = E \cup \{\overline{x_i \mapsto q_i}\}$
(^{2}) $\overline{q_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = be_i}\}$ **in** e y S. $E_1 = E \cup \{\overline{x_i \mapsto q_i}\}$
(^{3}) e_k se corresponde a la alternativa $C_k\ \overline{y_{ki}} \rightarrow e_k$ en $alts$
(^{4}) C_k no aparece en $alts$
(^{5}) $E_5 = \{x \mapsto p, \overline{x_i \mapsto p_i}^k\}$
(^{6}) $k_i =$ si $isInt(x_i)$ entonces x_i sino $E(x_i)$

Figura 5.2: La máquina abstracta *STG* modificada

Teniendo en cuenta la idea de que las nuevas expresiones pueden producir anotaciones, debemos escribir las reglas que traten las nuevas expresiones. En la Figura 5.3 se presentan las nuevas reglas de transición que modelan el comportamiento de dichas expresiones. Necesitaremos una nueva regla (*observer@*) que se encargará de la observación de los constructores. Además, será necesario realizar variaciones de las reglas *var1*, *var2* y *app2* de la máquina STG, para que tengan en cuenta las observaciones.

Recalcaremos que los efectos laterales se producen a la vez que el programa realiza su cómputo. Por tanto, las observaciones pueden ser revisadas incluso aunque el programa no finalice su

	Heap	Control	Entorno	Pila	Efecto lateral	Regla
	$H[q \mapsto closure]$	$x^{@str}$	$E\{x \mapsto q\}$	S	f	$var1@_1(^1)$
$\Rightarrow$	$H \cup [q' \mapsto closure]$	x	$\{x \mapsto q'^{@<str>}\}$	S	f	
	$H[q \mapsto closure]$	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S	f	$var1@_2(^1)$
$\Rightarrow$	$H \cup [q' \mapsto closure]$	x	$\{x \mapsto q'^{@<str +\!\!+ strs>}\}$	S	f	
	$H \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	S	f	$observer@(^2)$
$\Rightarrow$	K_1	x	$\{x \mapsto q\}$	S	$f \circ \langle strs, q \mapsto C_k\ \overline{p'_i} \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^n.e, E_1)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$	f	$app2@(^3)$
$\Rightarrow$	K_2	y	$\{y \mapsto q'^{@<strs>}\}$	S	$f \circ \langle strs, q \mapsto (\lambda \overline{p'_i}^n.q') \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$	f	$var2@_1(^4)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_3)]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs, q \mapsto \neg \overline{p_i}^k \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$	f	$var2@_2(^5)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_4)]$	x	$\{x \mapsto q^{@<strs +\!\!+ strs'>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs', q \mapsto \neg \overline{p_i}^k \rangle$	

(^{1}) q' es fresca con respecto a $H, x^{@str}$ y S.
(^{2}) $\overline{p'_i}$, q son distintas y frescas con respecto a H, x y S. $K_1 = H \cup [\overline{p'_i \mapsto H(p_i)}] \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i'^{@<strs>}}\})]$
(^{3}) $\overline{p'_i}$, q' son distintas y frescas con respecto a H, x y S. $K_2 = H \cup [\overline{p'_i \mapsto H(p_i)}] \cup [q' \mapsto (e, E_1 \cup \{\overline{x_i \mapsto p_i'^{@<strs>}}^n\})]$
(^{4}) $E_3 = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
(^{5}) $E_4 = \{x \mapsto q^{@<strs +\!\!+ strs'>}, \overline{x_i \mapsto p_i}^k\}$

Figura 5.3: Máquina abstracta *STG-Hood*

cómputo, ya que el fichero quedará en el sistema como efecto lateral.

Antes de describir el comportamiento de las reglas concretas, debemos introducir algo de notación. Debido a que permitimos anotar con una marca de observación cada estructura a ser observada, nuestros punteros también deben poder ser anotados con un conjunto de marcas de observaciones (la notación $< strs >$ representa un conjunto de *strings*). Debemos utilizar un conjunto de *strings* en lugar de un único *string* debido a que la misma clausura podría ser observada con diferentes marcas de observación en entornos distintos.

Por otro lado y debido a que ahora nuestros punteros pueden contener anotaciones de observaciones, la pila contendrá dos tipos distintos de punteros: los punteros "normales" y los punteros anotados con observaciones. Como ambos tipos de punteros se van a seguir considerando como punteros, las transiciones de la máquina abstracta se aplicarán sin diferencia sobre ambos tipos de punteros. Sin embargo, en ciertas ocasiones nos convendrá distinguirlos. Para realizar esto de una forma compacta, utilizaremos la siguiente notación: q hará referencia a punteros que no se encuentran bajo observación, mientras que cuando utilicemos p nos referiremos indistintamente a cualquier tipo de punteros (bajo observación o no). De esta manera evitaremos duplicar varias reglas.

Sin embargo, cuando utilicemos la notación $H \cup [p \mapsto closure]$ nos referiremos al puntero correspondiente a p, sin la marca de observación que pudiera tener p. Advertiremos que aunque los punteros del *heap* no tienen marcas de observación, los entornos asociados a las expresiones sí pueden tener punteros observados. Por ese motivo, cuando accedemos a una clausura puede que estemos tratando de entrar con un puntero bajo observación, aunque el *heap* sólo liga punteros sin observar a clausuras.

Pasaremos ahora a describir las principales diferencias con respecto a la máquina STG vista en el Capítulo 2.3. La primera de ellas es que cuando se entra en una variable bajo observación, la clausura correspondiente es clonada y entonces la clausura nueva es observada (esto se realiza

en las primeras dos reglas de la Figura 5.3). Es importante recalcar que en este caso estamos simulando el comportamiento que, según la documentación, posee Hood. En Hood, las clausuras bajo observación no son compartidas con otras partes del programa y, por tanto, duplican cálculos. De esta manera, es más sencillo definir quién fue el responsable de la evaluación de cada parte de la estructura.

Para comprender mejor el comportamiento de las reglas, asumiremos que $x^{@str}$ representa la clausura que está siendo observada bajo la marca de observación *str*, pero que aún no ha sido clonada. Es obvio que este tipo de clausuras sólo puede ser creada por el programador, es decir, una vez que el programa original ha comenzado su evaluación, no es posible que nuevas clausuras de este tipo aparezcan, ya sea debido a los efectos laterales de las transiciones de la máquina, o debido a la reducción de las expresiones. En contraposición, utilizaremos la notación $p^{@<strs>}$ para representar una clausura que ya ha sido clonada y que está siendo observada bajo el conjunto de marcas de observación *strs*; este tipo de clausuras surgen como resultado de la evolución del programa. En las situaciones donde una clausura esté siendo observada con varias marcas de observación será necesario realizar una concatenación de las marcas de observación. Será necesario definir una función que se encargue de concatenar la nueva marca de observación con el conjunto de marcas que estén observando la clausura. Utilizaremos el símbolo ($+\!\!+$) tanto para realizar la unión de dos conjuntos como para añadir un elemento a dicho conjunto.

A continuación pasaremos a describir el comportamiento de las reglas modificadas que se encargan de la evaluación de las nuevas expresiones.

- Las reglas *var1*$@_1$ y *var1*$@_2$ clonan la clausura observada y anotan el puntero que apunta a dicha clausura como observable. De esta forma se puede recordar de forma sencilla que la clausura ya ha sido clonada. Además, cuando una clausura se reduce a forma normal débil de cabeza (*whnf*), la máquina será capaz de detectar que se debe realizar el correspondiente efecto lateral para observar dicha *whnf*. Incluso en ese mismo momento será necesario clonar las clausuras referenciadas por dicha forma normal y anotarlas como observables.

  Nótese que debido a que es necesario modificar el entorno aprovechamos para realizar una poda sobre dicho entorno. De tal forma, sólo almacenamos la ligadura correspondiente a la variable de la expresión de control. Esto se realiza de la misma manera en todas las reglas de la Figura 5.3.

  La regla *var1*$@_1$ se encarga de la evaluación de una variable anotada como observable que no está previamente observada, véase que $(x \mapsto q)$. Mientras que la regla *var1*$@_2$ se encarga de la evaluación de una variable anotada como observable que ya está siendo observada, véase que $(x \mapsto q^{@<strs>})$, por tanto, lo único que hace es añadir la nueva marca de observación, *str*, a la lista de observaciones.

- La regla *observer*@ se corresponde con la observación de un constructor aplicado a su lista de argumentos. Cuando se obtiene el constructor, se produce un efecto lateral y se clonan las clausuras correspondientes con los argumentos de dicho constructor. Finalmente, dichas clausuras clonadas se marcan como observables.

  Desde ese momento, la evaluación continúa, pero debido a que ya se ha realizado la observación del constructor ya no es necesario mantener la observación de dicho constructor. Por este motivo la variable x del entorno queda vinculada al puntero q sin observación.

  Indicaremos también que la regla *observer*@ no modifica la pila, ya que sólo realiza una observación y no realiza ninguna evaluación.

- La regla *app2@* se utiliza para observar la aplicación completa, es decir, la aplicación de una función a todos los argumentos a los que está aplicada. Recordaremos que debido a ciertos detalles de eficiencia, la máquina STG sólo aplica las funciones a todos sus argumentos a la vez y nunca realiza aplicaciones parciales.

  Esta regla reduce la expresión $\lambda\overline{x_i}^n.e$ (donde $e \neq \lambda$) que está siendo observada bajo el conjunto de marcas de observación *strs*. Además, dicha función va a aplicarse a todos sus argumentos $\overline{p_i}^n$. Téngase en cuenta que, en esta situación, parece que lo razonable es observar tanto los argumentos de la función como el resultado de dicha aplicación, al igual que hace Hood. Por tanto, procedemos a observar ambas cosas:

  - clonamos las clausuras relativas a los argumentos marcando las nuevas como observables y
  - creamos una nueva clausura correspondiente al cuerpo de la lambda forma, aplicada a los nuevos argumentos, marcándola también como observable.

  Véase que en caso de que cualquiera de los argumentos p_i se encuentre bajo observación con otra marca de observación, entonces los correspondientes p'_i tendrán ambas marcas de observación, es decir, si $p'_i = q^{@<strs>}$ entonces $p'^{@<str>}_i = (q^{@<strs>})^{@<str>} = q^{@<str \mathbin{+\!\!+} strs>}$.

- Las reglas *var2*$@_1$ y *var2*$@_2$ se corresponden con actualizaciones de aplicaciones parciales. En este caso, la clausura q ha sido reducida a una aplicación parcial de sus primeros k parámetros de la λ-abstracción almacenada en la posición p del *heap*. En esta situación, se genera una observación inicial indicando que no se están observando los primeros k parámetros de dicha λ-abstracción y se continúa con la evaluación de dicha λ-abstracción, pero ahora marcada como observable. Este tipo de observación negativa podría haberse evitado si hubiéramos actualizado el *heap* en la regla *var2* con la clausura $q \mapsto (\lambda\overline{y_i}^n.e, E_1 \cup \{\overline{x_i \mapsto p_i}^k)\}|)$, ya que en este caso con crear una nueva clausura intermedia que se encargara de la observación no sería necesario crear la marca de observación negativa.

  Al crear la aplicación parcial estamos perdiendo la información relativa a que no estamos observando los argumentos que se encuentran almacenados en dicha aplicación parcial, por tanto, observaremos todos los argumentos de la λ-abstracción. Por ese motivo debemos indicar que los k argumentos no están bajo observación, por lo que introducimos las marcas negativas.

  Las diferencias entre ambas reglas *var2*$@_1$ y *var2*$@_2$ son las mismas que entre las reglas *var1*$@_1$ y *var1*$@_2$. En la primera la clausura no se encuentra previamente observada, mientras que en la segunda la clausura se encuentra ya bajo observación y, por tanto, es necesario juntar ambas marcas de observaciones.

Como puede verse, la observación de constructores es relativamente simple, ya que una vez observados no es necesario recordar que se encuentran bajo observación. Sin embargo, la observación de la aplicación de funciones es algo más complicada. De hecho, la observación de funciones se realiza a la misma vez que se evalúan a forma normal dichas aplicaciones, a través de las reglas *app2@*, *var2*$@_1$ y *var2*$@_2$, y no de forma independiente con la evaluación como lo hacen los constructores.

El comportamiento del depurador modelado en esta sección se corresponde con el comportamiento documentado de la librería Hood. Por este motivo lo denominaremos como *STG-Hood*.

Un postproceso adecuado del fichero de observaciones permite obtener las mismas observaciones que Hood.

Ejemplos de evaluación de STG-Hood

Veamos la evaluación de algunos ejemplos que, por un lado, nos clarifiquen el comportamiento de esta máquina y que, por otro lado, nos sean útiles para diferenciar las diferentes características de las versiones de Hood que iremos introduciendo en la máquina STG. Procuraremos mostrarlos de una forma sencilla y comprensible para el lector, de tal forma que queden patentes no sólo las diferencias, sino también la evaluación.

El primer ejemplo que mostraremos tiene que ver con los enteros. Nos interesa ver tanto la evaluación de dichos valores como las observaciones que sobre ellos se producen.

Ejemplo 5.1 A continuación pasaremos a presentar la expresión inicial de partida, primero en lenguaje Haskell y posteriormente la traducida a nuestro lenguaje STGL-Observado. En este ejemplo pretendemos observar el valor de un entero y la evaluación de la operación de suma que trabaja sobre enteros encapsulados. Ya vimos parte de los pasos que se producen al evaluar dicha operación cuando se trabaja sobre valores encapsulados en el Ejemplo 2.4, por tanto será más sencillo seguir los pasos de este ejemplo.

Presentamos la expresión inicial en el lenguaje Haskell:

```
observe "resul" (1 + (observe "intObs" 5))
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
  uno   = Int 1#
  cinco = Int 5#

  cinco0  = cinco@{intObs}

  res   = + uno cinco0
  res0  = res@{resul}
in res0
```

donde los nombres de las variables se han elegido apropiadamente con la intención de hacer el código más comprensible. Lógicamente, un generador automático de nombres crearía unos nombres de variables más ilegibles.

Véase que todos los enteros se han considerado enteros encapsulados y la operación de suma, por tanto, es la operación de suma sobre enteros encapsulados. En este caso, su evaluación es interesante ya que, como veremos, durante su cómputo se realizará la observación correspondiente a `intObs`. Normalmente, en los ejemplos no se mostrará la reducción de la operación suma, sino sólo su resultado, salvo que se considere interesante algún cómputo intermedio que ésta demande. Es más, en este ejemplo particular se considerará que dicha operación se corresponde con una clausura más en el *heap*, pero a partir de ahora ni siquiera se mostrará, se evaluará directamente.

Pasaremos ahora a ver la evaluación de dicha expresión en la máquina *STG-Hood*. Para hacerla más comprensible, en caso de que alguno de los componentes de la configuración no varíe, éste se mostrará con puntos suspensivos. Por otro lado, sólo resaltaremos los cambios que se produzcan en cualquiera de los componentes y el resto se mostrará también con puntos suspensivos.

Lo primero que se realiza es la eliminación del **letrec** y la demanda de la evaluación de la expresión `res0`, que a su vez provoca la demanda de la evaluación de la expresión `res`, que provocará la demanda de la suma. Cuando la suma se reduzca a forma normal se actualizarán las clausuras `res` y `res0`, en ese orden, y se producirá la anotación final de observación indicando el valor al que evaluó `res`. La evaluación que aparece a continuación se corresponde con la descrita en este párrafo. Posteriormente mostraremos la evaluación de la suma.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	$\{\}$	e_0	$\{\}$	$[\,]$	$\{\}$	*letrec*
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (cinco^{@\text{intObs}},\ \{cinco \mapsto p_2\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \\ p_5 \mapsto (res^{@\text{resul}},\ \{res \mapsto p_4\}) \end{Bmatrix}$	$resO$	$\begin{Bmatrix} uno \mapsto p_1 \\ cinco \mapsto p_2 \\ cincoO \mapsto p_3 \\ res \mapsto p_4 \\ resO \mapsto p_5 \end{Bmatrix}$	$[\,]$	$\{\}$	*var1*
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (cinco^{@\text{intObs}},\ \{cinco \mapsto p_2\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \end{Bmatrix}$	$res^{@\text{resul}}$	$\{res \mapsto p_4\}$	$[\#p_5]$	$\{\}$	$var1@_1$
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (cinco^{@\text{intObs}},\ \{cinco \mapsto p_2\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \\ p_6 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \end{Bmatrix}$	res	$\{res \mapsto p_6^{@<\text{resul}>}\}$	$[\#p_5]$	$\{\}$	*var1*
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (cinco^{@\text{intObs}},\ \{cinco \mapsto p_2\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \end{Bmatrix}$	$+\ uno\ cincoO$	$\begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}$	$[\#p_6^{@<\text{resul}>}, \#p_5]$	$\{\}$	*app1*
$\Longrightarrow^*$			...Reducción de la suma ...			
$\Longrightarrow$	$\begin{Bmatrix} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \begin{Bmatrix} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{Bmatrix}) \\ p_7 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_8 \mapsto (Int\ x,\ \{x \mapsto 6\#\}) \end{Bmatrix}$	sol	$\begin{Bmatrix} sol\# \mapsto 6\# \\ sol \mapsto p_8 \end{Bmatrix}$	$[\#p_6^{@<\text{resul}>}, \#p_5]$	$\{\ \ldots \text{Obs. suma} \ldots\ \}$	*var3*
$\Longrightarrow$	$\begin{Bmatrix} \ldots \\ p_6 \mapsto (Int\ x,\ \{x \mapsto 6\#\}) \end{Bmatrix}$	sol	$\{sol \mapsto p_6^{@<\text{resul}>}\}$	$[\#p_5]$	$\{\ \ldots \text{Obs. suma} \ldots\ \}$	*observer@*
$\Longrightarrow$	$\{\ldots\}$	sol	$\{sol \mapsto p_6\}$	$[\#p_5]$	$\begin{Bmatrix} \ldots \text{Obs. suma} \ldots \\ (\text{resul},\ p_6 \mapsto\ Int\ 6\#) \end{Bmatrix}$	*var3*
$\Longrightarrow$	$\begin{Bmatrix} \ldots \\ p_5 \mapsto (Int\ x,\ \{x \mapsto 6\#\}) \end{Bmatrix}$	sol	$\{sol \mapsto p_5\}$	$[\,]$	$\begin{Bmatrix} \ldots \text{Obs. suma} \ldots \\ (\text{resul},\ p_6 \mapsto\ Int\ 6\#) \end{Bmatrix}$	

Véase que, en este caso, cuando se aplica la regla *observer@* no se anota ninguna clausura como observable ya que es un valor entero primitivo. Es por ese motivo que se observa directamente el resultado del entero, no sólo del constructor, aún sin saber si va a ser demandado.

A continuación pasaremos a ver la evaluación de la suma y las modificaciones que provoca tanto sobre el *heap*, como sobre el fichero de observaciones:

	Heap	Control	Entorno	Pila	Efecto lateral	regla
$\Longrightarrow$	$\left\{\begin{array}{l} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (cinco^{@\text{intObs}},\ \{cinco \mapsto p_2\}) \end{array}\right\}$	$+\ uno\ cincoO$	$\left\{\begin{array}{l} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{array}\right\}$	S_I	$\{\}$	*app1*
$\Longrightarrow$	$\{\ldots\}$	$+$	$\left\{\begin{array}{l} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{array}\right\}$	$p_1 : p_3 : S_I$	$\{\}$	*app2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x_1\ \textbf{of}\ alts$	$\left\{\begin{array}{l} x_1 \mapsto p_1 \\ x_2 \mapsto p_3 \end{array}\right\}$	S_I	$\{\}$	*case1*
$\Longrightarrow$	$\{\ldots\}$	x_1	$\left\{\begin{array}{l} x_1 \mapsto p_1 \\ x_2 \mapsto p_3 \end{array}\right\}$	$(alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\}$	*case2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x_2\ \textbf{of}\ alts'$	$\left\{\begin{array}{l} x_2 \mapsto p_3 \\ x_1\# \mapsto 1\# \end{array}\right\}$	S_I	$\{\}$	*case1*
$\Longrightarrow$	$\{\ldots\}$	x_2	$\left\{\begin{array}{l} x_2 \mapsto p_3 \\ x_1\# \mapsto 1\# \end{array}\right\}$	$(alts',\ \{x_1\# \mapsto 1\#\}) : S_I$	$\{\}$	*var1*
$\Longrightarrow$	$\left\{\begin{array}{l} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \end{array}\right\}$	$cinco^{@\text{intObs}}$	$\{cinco \mapsto p_2\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\}$	$var1@_1$
$\Longrightarrow$	$\left\{\begin{array}{l} \ldots \\ p_7 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \end{array}\right\}$	$cinco$	$\left\{cinco \mapsto p_7^{@<\text{intObs}>}\right\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\}$	*observer@*
$\Longrightarrow$	$\{\ldots\}$	$cinco$	$\{cinco \mapsto p_7\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{(\text{intObs},\ p_7 \mapsto\ Int\ 5\#)\}$	*var3*
$\Longrightarrow$	$\left\{\begin{array}{l} \ldots \\ p_3 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \end{array}\right\}$	$cinco$	$\{cinco \mapsto p_3\}$	$(alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\ldots\}$	*case2*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x1\#\ +\#\ x2\#\ \textbf{of}\ alts''$	$\left\{\begin{array}{l} x_2 \mapsto p_3 \\ x_1\# \mapsto 1\# \\ x_2\# \mapsto 5\# \end{array}\right\}$	S_I	$\{\ldots\}$	*case1*
$\Longrightarrow$	$\{\ldots\}$	$x1\#\ +\#\ x2\#$	$\left\{\begin{array}{l} x_2 \mapsto p_3 \\ x_1\# \mapsto 1\# \\ x_2\# \mapsto 5\# \end{array}\right\}$	$(alts'',\ \{\}) : S_I$	$\{\ldots\}$	*op#*
$\Longrightarrow$	$\{\ldots\}$	$6\#$	$\{\}$	$(alts'',\ \{\}) : S_I$	$\{\ldots\}$	*case2v#*
$\Longrightarrow$	$\{\ldots\}$	$\textbf{letrec}\ sol = \ \ldots$	$\{sol\# \mapsto 6\#\}$	S_I	$\{\ldots\}$	*letrec*
$\Longrightarrow$	$\left\{\begin{array}{l} p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\}) \\ p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_3 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_4 \mapsto (+\ uno\ cincoO,\ \left\{\begin{array}{l} + \mapsto p_0 \\ uno \mapsto p_1 \\ cincoO \mapsto p_3 \end{array}\right\}) \\ p_7 \mapsto (Int\ x,\ \{x \mapsto 5\#\}) \\ p_8 \mapsto (Int\ x,\ \{x \mapsto 6\#\}) \end{array}\right\}$	sol	$\left\{\begin{array}{l} sol\# \mapsto 6\# \\ sol \mapsto p_8 \end{array}\right\}$	S_I	$\{(\text{intObs},\ p_7 \mapsto\ Int\ 5\#)\}$	

Una de las cosas interesantes es que la clausura correspondiente al puntero p_4 no ha sido reducida. Esto es debido a que Hood primero duplica la clausura. Si esta clausura fuera demandada desde otro punto de cómputo se tendría que reducir a *whnf*. Sin embargo su clausura clonada, es decir, la referente al puntero p_6, ya se ha reducido a *whnf*, es decir, los cálculos no se comparten.

Lo más interesante a resaltar en este cómputo es que la observación del entero *5* no sólo nos lleva a observar el constructor, sino que también observamos directamente el entero primitivo. Esto aparentemente se podría considerar un error, pero dicho valor ya está calculado y, por tanto, no es absurdo observarlo. Como comentamos en el Sección 3.4.2, éste es el comportamiento original de Hood con respecto a los valores básicos.

A nivel de la máquina STG nos podemos plantear otras opciones. Otra alternativa consiste en observar sólo el constructor y, por tanto, generar la observación sólo del constructor *Int* para posteriormente observar el valor del entero que encapsulaba, es decir, queremos obtener una

observación similar a ésta:

$$\left\{\begin{array}{l}(\text{intObs},\ p_7 \mapsto\ Int\ p_7\#)\\(\text{intObs},\ p_7\# \mapsto\ 5\#)\end{array}\right\}$$

La observación del entero desencapsulado debe ser realizada cuando dicho entero se utilice, es decir, cuando se consulte su valor en el entorno, por tanto, al realizar la operación primitiva de suma. Hasta el momento en que se realice la observación será necesario recordar que ese entero se está observando. Al igual que hicimos con los punteros habrá que anotar el entero con la marca de observación @ < intObs >. De esta manera aparece un nuevo tipo de datos, que son los valores primitivos observados. Esto conlleva a que las operaciones primitivas sean capaces de trabajar con este nuevo tipo de datos y producir las observaciones correspondientes en caso de que estén siendo observados. Aún así, la mayor dificultad radica en cómo enganchar la observación del entero desencapsulado con la correspondiente observación del constructor *Int*, su padre. La forma más simple que se nos ocurre es añadir en la marca de observación quién es el padre. En nuestro ejemplo la nueva marca de observación que se añadirá al entero será @ < $\text{intObs}_{p_7\#}$ >.

La evaluación que se presenta a continuación trata de modelar el comportamiento descrito anteriormente. Las reglas modificadas han sido anotadas con una prima.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
$\Longrightarrow$	$\left\{\begin{array}{l}p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\})\\p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\})\\p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\})\end{array}\right\}$	$cinco^{@\text{intObs}}$	$\{cinco \mapsto p_2\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\}$	$var1@_1$'
$\Longrightarrow$	$\left\{p_7 \mapsto (Int\ x,\ \left\{x \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\right\})\right\}$	$cinco$	$\left\{cinco \mapsto p_7^{@<\text{intObs}>}\right\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\}$	$observer@$
$\Longrightarrow$	$\{\ldots\}$	$cinco$	$\{cinco \mapsto p_7\}$	$\#p_3 : (alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{(\text{intObs},\ p_7 \mapsto\ Int\ p_7\#)\}$	$var3$
$\Longrightarrow$	$\left\{p_3 \mapsto (Int\ x,\ \left\{x \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\right\})\right\}$	$cinco$	$\{cinco \mapsto p_3\}$	$(alts,\ \{x_2 \mapsto p_3\}) : S_I$	$\{\ldots\}$	$case2$
$\Longrightarrow$	$\{\ldots\}$	$\textbf{case}\ x1\#\ +\#\ x2\#\ \textbf{of}\ alts''$	$\left\{\begin{array}{l}x_2 \mapsto p_3\\x_1\# \mapsto 1\#\\x_2\# \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\end{array}\right\}$	S_I	$\{\ldots\}$	$case1$
$\Longrightarrow$	$\{\ldots\}$	$x1\#\ +\#\ x2\#$	$\left\{\begin{array}{l}x_2 \mapsto p_3\\x_1\# \mapsto 1\#\\x_2\# \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\end{array}\right\}$	$(alts'',\ \{\}) : S_I$	$\{\ldots\}$	$op\#$'
$\Longrightarrow$	$\{\ldots\}$	$6\#$	$\{\}$	$(alts'',\ \{\}) : S_I$	$\left\{\begin{array}{l}\ldots\\(\text{intObs},\ p_7\# \mapsto\ 5\#)\end{array}\right\}$	$case2v\#$
$\Longrightarrow$	$\{\ldots\}$	$\textbf{letrec}\ sol = \ \ldots$	$\{sol\# \mapsto 6\#\}$	S_I	$\{\ldots\}$	$letrec$
$\Longrightarrow$	$\left\{\begin{array}{l}p_0 \mapsto (\lambda\ x_1\ x_2.\ \textbf{case}\ \ldots,\ \{\})\\p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\})\\p_2 \mapsto (Int\ x,\ \{x \mapsto 5\#\})\\p_3 \mapsto (Int\ x,\ \left\{x \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\right\})\\p_4 \mapsto (+\ uno\ cincoO,\ \left\{\begin{array}{l}+ \mapsto p_0\\uno \mapsto p_1\\cincoO \mapsto p_3\end{array}\right\})\\p_7 \mapsto (Int\ x,\ \left\{x \mapsto 5\#^{@<\text{intObs}_{p_7\#}>}\right\})\\p_8 \mapsto (Int\ x,\ \{x \mapsto 6\#\})\end{array}\right\}$	sol	$\left\{\begin{array}{l}sol\# \mapsto 6\#\\sol \mapsto p_8\end{array}\right\}$	S_I	$\left\{\begin{array}{l}(\text{intObs},\ p_7 \mapsto\ Int\ p_7\#)\\(\text{intObs},\ p_7\# \mapsto\ 5\#)\end{array}\right\}$	

Como se puede observar en este ejemplo, a nivel de la máquina STG se poseen muchas opciones. Las consecuencias de tratar de intentar realizar las observaciones sobre los enteros desencapsulados de esta última forma son las siguientes:

- añadir un nuevo tipo de valores primitivo, sobre cada valor primitivo, los primitivos observados

- modificar las reglas $var1@_1$ y $op\#$ para que trabajen adecuadamente con el nuevo tipo de datos

En el momento de la codificación de dicha máquina en una arquitectura real, estas modificaciones nos llevarían a muchos problemas, sobre todo de ineficiencia. Por este motivo, a nivel de la máquina STG nos quedaremos con la primera versión de tratamiento de los enteros desencapsulados. Dicha versión se corresponde con el comportamiento de la librería Hood. □

Pasaremos ahora a ver un segundo ejemplo. Dicho ejemplo es algo más complicado, ya que se centra en la evaluación y observación de una función y servirá para mostrar las diferencias entre las distintas versiones de depuración en la máquina *STG* que presentaremos en este capítulo.

Ejemplo 5.2 A continuación pasaremos a presentar la expresión inicial de partida, primero en lenguaje Haskell y posteriormente la traducida a nuestro lenguaje STGL-Observado. En este ejemplo pretendemos observar una función. Más concretamente observaremos la función `length` aplicada a una lista compuesta por un único número entero. Ahora bien, lo haremos en un contexto en que los elementos de la lista son demandados por una segunda función no observada. Para ello, sumaremos el resultado de la longitud de la lista con la suma de los valores de la lista. Esto último será interesante cuando queramos ver las distintas observaciones que se obtienen con otras versiones de depuración en la máquina *STG*.

Aunque las funciones `length` y `sum` ya están previamente definidas en el preludio de Haskell, nosotros las mostraremos aquí por completitud. Partimos, por tanto, de la siguiente expresión inicial de Haskell:

```
(observe "length" length xs) + (sum xs)
  where
    xs = 2:[]

    length (x:xs) = 1 + length xs
    length []     = 0

    sum (x:xs) = x + sum xs
    sum []     = 0
```

que en nuestro lenguaje tras el proceso de normalización y varias optimizaciones se convierte en la siguiente expresión de partida e_0:

```
letrec
  cero   = Int 0#
  uno    = Int 1#
  dos    = Int 2#

  list  = CONS dos list1
  list1 = NIL

  length = \xs. case xs of
                  CONS y ys -> case length ys of
                                 sol -> + uno sol
                  NIL     -> cero

  sum = \xs. case xs of
                  CONS y ys -> case sum ys of
```

```
                                    sol -> + y sol
                  NIL      -> cero

  length0 = length@length

  x1 = length0 list
  x2 = sum list
in + x1 x2
```

Dése cuenta que se ha considerado lo siguiente:

- la lista `xs` es una lista de enteros encapsulados (`xs :: [Int]`),
- la función `length` es una función que dado una lista devuelve un entero encapsulado (`length :: [a] -> Int`)
- la función `sum` es una función que recibe una lista de enteros encapsulados y devuelve un entero encapsulado (`sum :: [Int] -> Int`)
- el tipo de la expresión final se corresponde también con un entero encapsulado.

Pasaremos ahora a ver la evaluación de dicha expresión en la máquina *STG-Hood*. Para que se pueda seguir de una forma sencilla la evaluación, utilizaremos la notación vista en el ejemplo anterior. En este caso, las reducciones de la operación de suma, no primitiva, no se mostrarán por ser todas ellas equivalentes, sencillas y carentes de interés.

El primer paso de la máquina se corresponde con la reducción del **letrec**, que produce el siguiente *heap*:

$$\left\{\begin{array}{l}
p_0 \mapsto (Int\ x,\ \{x \mapsto 0\#\}) \\
p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\
p_2 \mapsto (Int\ x,\ \{x \mapsto 2\#\}) \\
p_3 \mapsto (Cons\ dos\ list1,\ \left\{\begin{array}{l} dos \mapsto p_2 \\ list1 \mapsto p_4 \end{array}\right\}) \\
p_4 \mapsto (NIL,\ \{\}) \\
p_5 \mapsto (\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad CONS\ y\ ys{-}{>} \\
\qquad\quad \textbf{case}\ length\ ys\ \textbf{of} \\
\qquad\qquad sol{-}{>} + uno\ sol \\
\qquad NIL{-}{>} cero \\
\qquad ,\ \left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}) \\
p_6 \mapsto (\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad CONS\ y\ ys{-}{>} \\
\qquad\quad \textbf{case}\ sum\ ys\ \textbf{of} \\
\qquad\qquad sol{-}{>} + y\ sol \\
\qquad NIL{-}{>} cero \\
\qquad ,\ \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) \\
p_7 \mapsto (length^{@length},\ \{length \mapsto p_5\}) \\
p_8 \mapsto (lengthO\ list,\ \left\{\begin{array}{l} lentghO \mapsto p_7 \\ list \mapsto p_3 \end{array}\right\}) \\
p_9 \mapsto (sum\ list,\ \left\{\begin{array}{l} sum \mapsto p_6 \\ list \mapsto p_3 \end{array}\right\})
\end{array}\right\}$$

Esto provoca que la expresión de control sea la suma. Su evaluación demanda la reducción de sus dos argumentos. Veamos en este caso únicamente la reducción de su primer argumento, `lengthO list`, que es la que provocará las anotaciones en el fichero.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h_I	*lengthO list*	$\{lentghO \mapsto p_7, list \mapsto p_3\}$	S_I	$\{\}$	*app1*
$\Longrightarrow$	h_I	*lengthO*	$\{lentghO \mapsto p_7, list \mapsto p_3\}$	$p_3 : S_I$	$\{\}$	*var1*
$\Longrightarrow$	$h_I - p_7$	$length^{@length}$	$\{length \mapsto p_5\}$	$\#p_7 : p_3 : S_I$	$\{\}$	$var1@_1$
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\})\}$	*length*	$\{length \mapsto p_{10}^{@<length>}\}$	$\#p_7 : p_3 : S_I$	$\{\}$	*var2*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\})\}$	*length*	$\{length \mapsto p_{10}^{@<length>}\}$	$p_3 : S_I$	$\{\}$	*app2@*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_2, list1 \mapsto p_4\}),\ p_{12} \mapsto (\mathbf{case}\ xs\ \ldots,\ \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, xs \mapsto p_{11}^{@<length>}\})\}$	*length*	$\{length \mapsto p_{12}^{@<length>}\}$	S_I	$\{(\text{length},\ p_{10} \mapsto\ \lambda\, p_{11}.\, p_{12})\}$	*var1*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_2, list1 \mapsto p_4\})\}$	**case** *xs* **of** *alts*	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, xs \mapsto p_{11}^{@<length>}\}$	$\#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto\ \lambda\, p_{11}.\, p_{12})\}$	*case1*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_2, list1 \mapsto p_4\})\}$	*xs*	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, xs \mapsto p_{11}^{@<length>}\}$	$(alts,\ \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}) : \#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto\ \lambda\, p_{11}.\, p_{12})\}$	*observer@*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_{13}^{@<length>}, list1 \mapsto p_{14}^{@<length>}\}),\ p_{13} \mapsto (Int\ x, \{x \mapsto 2\#\}),\ p_{14} \mapsto (Nil, \{\})\}$	*xs*	$\{xs \mapsto p_{11}\}$	$(alts,\ \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}) : \#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto \lambda\, p_{11}.\, p_{12}), (\text{length},\ p_{11} \mapsto Cons\ p_{13}\ p_{14})\}$	*case2*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_{13}^{@<length>}, list1 \mapsto p_{14}^{@<length>}\}),\ p_{13} \mapsto (Int\ x, \{x \mapsto 2\#\}),\ p_{14} \mapsto (Nil, \{\})\}$	**case** *length ys* ...	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, ys \mapsto p_{14}^{@<length>}\}$	$\#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto \lambda\, p_{11}.\, p_{12}), (\text{length},\ p_{11} \mapsto Cons\ p_{13}\ p_{14})\}$	*case1*
$\Longrightarrow$	$\{h_I - p_7,\ p_{10} \mapsto (\lambda xs.\ \ldots,\ \{\ldots\}),\ p_7 \mapsto (length,\ \{length \mapsto p_{10}^{@<length>}\}),\ p_{11} \mapsto (Cons\ dos\ list1,\ \{dos \mapsto p_{13}^{@<length>}, list1 \mapsto p_{14}^{@<length>}\}),\ p_{13} \mapsto (Int\ x, \{x \mapsto 2\#\}),\ p_{14} \mapsto (Nil, \{\})\}$	*length ys*	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, ys \mapsto p_{14}^{@<length>}\}$	$(alts',\ \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}) : \#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto \lambda\, p_{11}.\, p_{12}), (\text{length},\ p_{11} \mapsto Cons\ p_{13}\ p_{14})\}$	*app1*
$\Longrightarrow^*$		...Reducción de la llamada recursiva...				
$\Longrightarrow$	$\{h_I'' - p_{14},\ p_{14} \mapsto (Nil,\ \{\})\}$	*cero*	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}$	$(alts',\ \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}) : \#p_{12}^{@<length>} : S_I$	$\{(\text{length},\ p_{10} \mapsto\ \lambda\, p_{11}.\, p_{12}), (\text{length},\ p_{11} \mapsto Cons\ p_{13}\ p_{14}), (\text{length},\ p_{14} \mapsto\ Nil)\}$	*case2*
$\Longrightarrow$	$\{h_I'' - p_{14},\ p_{14} \mapsto (Nil,\ \{\})\}$	+ *uno sol*	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, sol \mapsto p_0\}$	$\#p_{12}^{@<length>} : S_I$	$\{\ldots\}$	
$\Longrightarrow^*$		...Reducción de la suma...				
$\Longrightarrow$	$\{\ldots,\ p_{15} \mapsto (Int\ x,\ \{x \mapsto 1\#\})\}$	*sol*	$\{sol \mapsto p_{15}\}$	$\#p_{12}^{@<length>} : S_I$	$\{\ldots\}$	*var3*
$\Longrightarrow$	$\{\ldots,\ p_{15} \mapsto (Int\ x,\ \{x \mapsto 1\#\})\}$	*sol*	$\{sol \mapsto p_{12}^{@<length>}\}$	S_I	$\{\ldots\}$	*observer@*
$\Longrightarrow$	$\{\ldots,\ p_{15} \mapsto (Int\ x,\ \{x \mapsto 1\#\})\}$	*sol*	$\{sol \mapsto p_{12}\}$	S_I	$\{\ldots, (\text{length},\ p_{12} \mapsto\ \{x \mapsto 1\#\})\}$	

A continuación pasaremos a ver la evaluación de la llamada recursiva `length list1`, que primero provocará modificaciones sobre el *heap*, luego generará otra observación y finalmente nos devolverá el valor entero `cero`:

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h'_I	*length ys*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ ys \mapsto p_{14}^{@<length>} \end{array}\right\}$	S'_I	f_I	*app1*
$\Longrightarrow$	h'_I	*length*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ ys \mapsto p_{14}^{@<length>} \end{array}\right\}$	$p_{14}^{@<length>} : S'_I$	f_I	*app2*
$\Longrightarrow$	h'_I	**case** *xs* **of** *alts*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ xs \mapsto p_{14}^{@<length>} \end{array}\right\}$	S'_I	f_I	*case1*
$\Longrightarrow$	h'_I	*xs*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ xs \mapsto p_{14}^{@<length>} \end{array}\right\}$	$(alts, \left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}) : S'_I$	f_I	*observer@*
$\Longrightarrow$	h'_I	*xs*	$\{xs \mapsto p_{14}\}$	$(alts, \left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}) : S'_I$	$\left\{\begin{array}{l} f_I \\ (\text{length},\ p_{14} \mapsto\ Nil) \end{array}\right\}$	*case2*
$\Longrightarrow$	h'_I	*cero*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}$	S'_I	$\left\{\begin{array}{l} f_I \\ (\text{length},\ p_{14} \mapsto\ Nil) \end{array}\right\}$	

A partir de este punto, se comenzaría a evaluar el segundo argumento de la suma, es decir, la clausura p_9 `(sum list)`. En este caso la variable `list` hace referencia a la clausura p_3 que no ha sido modificada por las observaciones, es decir, no posee ninguna marca de observación. Es por ese motivo que la reducción de dicha clausura no generará ninguna observación. Por tanto, el fichero de observaciones al finalizar el cómputo contendría las siguientes observaciones:

$$\left\{\begin{array}{l} (\text{length},\ p_{10} \mapsto\ \lambda\ p_{11}.\ p_{12}) \\ (\text{length},\ p_{11} \mapsto Cons\ p_{13}\ p_{14}) \\ (\text{length},\ p_{14} \mapsto\ Nil) \\ (\text{length},\ p_{12} \mapsto\ \{x \mapsto 1\#\}) \end{array}\right\}$$

que se mostrarían al usuario siguiendo el estilo de las observaciones de Hood, es decir, se mostraría el siguiente mensaje de texto:

```
-- length
     \ (_:[]) -> 1
```

□

5.2.2. Compartiendo clausuras: depurador similar a Hugs-Hood

Tal y como se ha visto en la versión anterior, cuando se está observando una estructura se pierde la compartición de las clausuras. Esto implica que se necesita más memoria para la ejecución del programa y también implica que algunos cómputos se dupliquen. Por tanto, se reduce la eficiencia tanto en el tiempo de ejecución como en el espacio.

Como se ha comentado en el Capítulo 3.4, el intérprete Hugs incluye una versión diferente de Hood. Una de las ventajas principales de dicha versión es que no se pierde la compartición de las clausuras cuando éstas se encuentran bajo observación. Sin embargo, los resultados obtenidos con dicha versión de Hood son sustancialmente diferentes de los resultados que se obtienen con la librería original. La diferencia consiste en que Hugs no almacena la información relativa a quién es el responsable de la evaluación de la clausura. Sólo almacena la información relativa a si un valor ha sido computado o no.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	H	$x^{@str}$	$E\{x \mapsto q\}$	S	f	$var1@_1$
$\Rightarrow$	H	x	$\{x \mapsto q^{@<str>}\}$	S	f	
	H	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S	f	$var1@_2$
$\Rightarrow$	H	x	$\{x \mapsto q^{@<str +\!\!+ strs>}\}$	S	f	
	$H \cup [q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	S	f	$observerH@$
$\Rightarrow$	$H \cup [q \mapsto (C\ \overline{x_i}, \{x_i \mapsto p_i^{@<strs>}\})]$	x	$\{x \mapsto q\}$	S	$f \circ \langle strs, q \mapsto C\ \overline{p_i} \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^n.e, E_1)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$	f	$app2@$
$\Rightarrow$	$H \cup [q' \mapsto (e, E_1 \cup \{x_i \mapsto p_i^{@<strs>}{}^n\})]$	x	$\{x \mapsto q'^{@<strs>}\}$	S	$f \circ \langle strs, q \mapsto (\lambda \overline{p_i}^n.q') \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E_0)]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$	f	$var2@_1(^1)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_2)]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs, q \mapsto \neg \overline{p_i}^k \rangle$	
	$H[q \mapsto (\lambda \overline{x_i}^k.\lambda \overline{y_i}^n.e, E_0)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$	f	$var2@_2(^2)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_2)]$	x	$\{x \mapsto q^{@<strs +\!\!+ strs'>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs', q \mapsto \neg \overline{p_i}^k \rangle$	

$(^1)$ $E_1 = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
$(^2)$ $E_2 = \{x \mapsto q^{@<strs +\!\!+ strs'>}, \overline{x_i \mapsto p_i}^k\}$

Figura 5.4: Máquina abstracta Hugs-Hood

En este caso modelaremos una depuración que mantenga las clausuras compartidas, aunque con ello se pierda la información relativa a quién ha realizado la observación. Llamaremos a dicha variante *Hugs-Hood* ya que esta versión está inspirada en el comportamiento de la librería de Hugs, aunque en algunos casos no producirá las mismas observaciones que dicha librería. Para ello, veremos que sólo será necesario una leve modificación de las reglas que se han introducido en la sección anterior. Las nuevas reglas se encuentran en la Figura 5.4. Véase que las reglas $var1@_1$ y $var1@_2$ son esencialmente las mismas que en la versión anterior, la principal diferencia es que ahora no se clona la clausura. La regla $observerH@$ también ha sido modificada para eliminar la duplicación de la clausura bajo evaluación. Eliminando dichas duplicaciones mantenemos la compartición de las clausuras, tal y como hace la librería de Hugs. Como contrapartida, hemos de decir que de esta manera se pierde la información de qué clausura está observando dicha clausura y se considerará que cualquier clausura que entre a demandar la evaluación de dicha clausura está observando su resultado.

Tenemos que tener en cuenta que también es necesario realizar una modificación sobre la regla $app2@$ evitando que realice la clonación de los parámetros de la aplicación. Ahora bien, sigue siendo necesario utilizar un nuevo puntero para almacenar el cuerpo de la λ-abstracción. Esta ligadura se crea para poder observar el resultado de la aplicación. En esta regla no se clonan las clausuras, por lo que también se mantiene la compartición.

Finalmente, las reglas $var2@_1$ y $var2@_2$ no han sufrido ningún cambio, es decir, son exactamente las mismas que en el modelo anterior. La razón para ello es que la única diferencia con respecto a la implementación anterior es que ahora no queremos clonar las clausuras. Sin embargo, nos alejamos de la idea de observar sólo lo demandado por la función. El motivo por el que estamos perdiendo la información relativa a la demanda de las clausuras es que ahora estamos marcando todas las clausuras que están bajo observación, sin preocuparnos de qué clausuras están observándolas.

Ejemplo de evaluación de Hugs-Hood

Para entender mejor la diferencia de comportamiento entre ambas versiones veamos sobre el mismo ejemplo cómo se reducen las expresiones y qué observaciones se obtienen sobre el mismo ejemplo que en el caso anterior.

Ejemplo 5.3 Tanto la expresión Haskell como la expresión transformada son las mismas que en el Ejemplo 5.2. Por este motivo no las presentaremos aquí.

Como se vio, la reducción de esta expresión provocaba que se demandara la reducción de `x1` que, en este caso, provocará las mismas observaciones que en la versión STG-Hood. La diferencia radica en el hecho de que ahora no se duplican las clausuras. Por tanto, el *heap* que se obtiene tras la reducción de `x1` es el siguiente:

$$\left\{\begin{array}{l} p_0 \mapsto (Int\ x, \{x \mapsto 0\#\}) \\ p_1 \mapsto (Int\ x, \{x \mapsto 1\#\}) \\ p_2 \mapsto (Int\ x, \{x \mapsto 2\#\}) \\ p_3 \mapsto (Cons\ dos\ list1, \left\{\begin{array}{l} dos \mapsto p_2^{@<length>} \\ list1 \mapsto p_4^{@<length>} \end{array}\right\}) \\ p_4 \mapsto (Nil\ , \{\}) \\ p_5 \mapsto (\lambda xs.\textbf{case}\ xs\ \ldots, \left\{\begin{array}{l} cero \mapsto p_0 \\ length \mapsto p_5 \\ uno \mapsto p_1 \end{array}\right\}) \\ p_6 \mapsto (\lambda xs.\textbf{case}\ xs\ \ldots, \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) \\ p_7 \mapsto (length, \left\{length \mapsto p_5^{@<length>}\right\}) \\ p_8 \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \\ p_9 \mapsto (sum\ list, \left\{\begin{array}{l} list \mapsto p_3 \\ sum \mapsto p_6 \end{array}\right\}) \\ p_{10} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \\ p_{11} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \end{array}\right\}$$

y, por tanto, el fichero de observaciones que se obtiene es:

$$\left\{\begin{array}{l} (\text{length},\ p_5 \mapsto\ \lambda\ p_3.\ p_{10}) \\ (\text{length},\ p_3 \mapsto Cons\ p_2\ p_4) \\ (\text{length},\ p_4 \mapsto\ Nil) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\}) \end{array}\right\}$$

Tras la reducción del primer argumento de la suma, se provoca la reducción del segundo argumento de la suma, que se corresponde con el cómputo de la expresión `sum list`. Esto provocará por un lado una llamada recursiva a la función `sum` sobre la lista `Nil`. Esta llamada recursiva provocará una anotación en el fichero indicando que hemos evaluado la lista `Nil` (la mostraremos por separado). Cuando dicha llamada recursiva finalice, se realizará el cómputo correspondiente a `+ y sol` que demandará la evaluación de sus argumentos y realizará una nueva anotación en el fichero de observaciones. De esta suma sólo mostraremos los pasos hasta que se produzca la anotación de observación.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h_I	$sum\ list$	$\left\{\begin{array}{l} list \mapsto p_3 \\ sum \mapsto p_6 \end{array}\right\}$	S_I	$\{\ \ldots \text{Obs. length} \ldots\ \}$	$app1$
$\Longrightarrow$	h_I	sum	$\left\{\begin{array}{l} list \mapsto p_3 \\ sum \mapsto p_6 \end{array}\right\}$	$p_3 : S_I$	$\{\ldots\}$	$app2$
$\Longrightarrow$	h_I	**case** xs **of** $alts$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ xs \mapsto p_3 \end{array}\right\}$	S_I	$\{\ldots\}$	$case1$
$\Longrightarrow$	h_I	xs	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ xs \mapsto p_3 \end{array}\right\}$	$(alts, \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) : S_I$	$\{\ldots\}$	$case2$
$\Longrightarrow$	h_I	**case** $sum\ ys$ **of** $alts'$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	S_I	$\{\ldots\}$	$case1$
$\Longrightarrow$	h_I	$sum\ ys$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	$(alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\{\ldots\}$	$app1$
$\Longrightarrow^*$			$\ldots$ Reducción de la llamada recursiva $\ldots$			
$\Longrightarrow$	h_I	$+\ y\ sol$	$\left\{\begin{array}{l} sol \mapsto p_0 \\ y \mapsto p_2^{@<length>} \end{array}\right\}$	S_I	$\{\ldots\}$	$app1$
$\Longrightarrow$	h_I	$+$	$\left\{\begin{array}{l} sol \mapsto p_0 \\ y \mapsto p_2^{@<length>} \end{array}\right\}$	$p_2^{@<length>} : p_0 : S_I$	$\left\{\begin{array}{l} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \end{array}\right\}$	$app2$
$\Longrightarrow$	h_I	**case** $x1$ **of** $alts''$	$\left\{\begin{array}{l} x1 \mapsto p_2^{@<length>} \\ x2 \mapsto p_0 \end{array}\right\}$	S_I	$\{\ldots\}$	$case1$
$\Longrightarrow$	h_I	$x1$	$\left\{\begin{array}{l} x1 \mapsto p_2^{@<length>} \\ x2 \mapsto p_0 \end{array}\right\}$	$(alts'', \{x2 \mapsto p_0\}) : S_I$	$\{\ldots\}$	$observerH@$
$\Longrightarrow$	h_I	$x1$	$\{x1 \mapsto p_2\}$	$(alts'', \{x2 \mapsto p_0\}) : S_I$	$\left\{\begin{array}{c} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\}) \end{array}\right\}$	$case2$
$\Longrightarrow^*$			$\ldots$ Reducción del resto de la suma $\ldots$			
$\Longrightarrow$	$\left\{\begin{array}{c} \ldots h_I \ldots \\ p_{12} \mapsto (Int\ sol\#, \{sol\# \mapsto 2\#\}) \end{array}\right\}$	sol	$\{sol \mapsto p_{12}\}$	S_I	$\left\{\begin{array}{c} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\}) \end{array}\right\}$	$\ldots$

Veamos ahora el cómputo de la llamada recursiva `sum list1`, que provocará la observación de la lista vacía en el fichero de observaciones.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h_I	$sum\ ys$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	$(alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\{\ldots\}$	$app1$
$\Longrightarrow$	h_I	sum	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	$p_4^{@<length>} : (alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\{\ldots\}$	$app2$
$\Longrightarrow$	h_I	**case** xs **of** $alts$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ xs \mapsto p_4^{@<length>} \end{array}\right\}$	$(alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\{\ldots\}$	$case1$
$\Longrightarrow$	h_I	xs	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \\ xs \mapsto p_4^{@<length>} \end{array}\right\}$	$(alts, \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) : (alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\{\ldots\}$	$observerH@$
$\Longrightarrow$	h_I	xs	$\{xs \mapsto p_4\}$	$(alts, \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) : (alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\left\{\begin{array}{c} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \end{array}\right\}$	$case2$
$\Longrightarrow$	h_I	$cero$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}$	$(alts', \left\{y \mapsto p_2^{@<length>}\right\}) : S_I$	$\left\{\begin{array}{c} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \end{array}\right\}$	$case2v$

Como se ha mostrado, la reducción de la expresión completa provoca la observación de los valores de la lista. Aunque la función `length` sólo haya demandado la espina de `xs`, realmente la

lista `xs` se evalúa de forma completa debido a la función `sum`. Las observaciones que se obtienen en el fichero se corresponden con:

$$\left\{\begin{array}{l}(\text{length},\ p_5 \mapsto\ \lambda\ p_3.\ p_{10}) \\ (\text{length},\ p_3 \mapsto\ \mathit{Cons}\ p_2\ p_4) \\ (\text{length},\ p_4 \mapsto\ \mathit{Nil}) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\}) \\ (\text{length},\ p_4 \mapsto\ \mathit{Nil}) \\ (\text{length},\ p_2 \mapsto\ \{x \mapsto 2\#\})\end{array}\right\}$$

que se mostraría al usuario de la siguiente forma:

```
-- length
   { \ (2:[]) -> 1 }
```

□

Como se ha comentado previamente, en este ejemplo se obtiene exactamente la misma observación que la que se obtendría con la librería de Hugs. La razón es que la máquina abstracta, a la hora de realizar el cómputo de las expresiones, no está preocupándose de quién es el observador de la clausura observada, es decir, en caso de que una clausura bajo observación sea demandada, su evaluación será observada incluso si la reducción de dicha clausura no se debe a la función que la está observando.

No obstante, en esta versión el orden de evaluación influye en las observaciones que se obtienen. Ya que lo que se observa se corresponde únicamente con cada una de las veces que se entra en una clausura que está marcada como observable. Por ese motivo si en vez de la expresión del ejemplo tuviéramos `(sum xs) + (observe "length" length xs)` se observaría sólo lo que la función `length` demande, ya que primero se evalúa la suma, que en este caso no realizaría ninguna observación y posteriormente se evaluaría `observe "length" length xs` que realizaría las observaciones. Por tanto, en este caso obtendríamos las mismas observaciones que con la librería Hood. La idea intuitiva es que se observa sólo a partir de que alguien indica que quiere observar dicha estructura, no antes.

5.2.3. Hood compartiendo clausuras

En la Sección 5.2.1 introdujimos en la máquina abstracta *STG* un comportamiento similar al de Hood. Desafortunadamente, con esta primera aproximación se perdía la compartición de las clausuras, dando lugar a una máquina más ineficiente en tiempo y espacio.

En la sección anterior hemos visto otra implementación que mantiene la compartición de las clausuras. Pero, sin embargo, en varias situaciones su comportamiento no es equivalente al que se obtiene con la versión original de Hood.

Afortunadamente, cuando se trabaja al nivel de las máquinas abstractas, ciertos problemas y optimizaciones son más sencillos de llevar a cabo. De hecho, en esta sección mostraremos cómo se puede mantener el comportamiento de la depuración de *STG-Hood* sin reducir la compartición de las clausuras, es decir, los programas observados se ejecutarán en un espacio un poco mayor que los no observados, debido a los **letrec** extras que es necesario añadir, pero mantendrán la compartición de cómputos y nunca evaluarán dos veces la misma clausura.

En la Figura 5.5 se presentan las nuevas transiciones necesarias para modelar el comportamiento de las observaciones manteniendo la compartición de las clausuras. Antes de entrar a

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	H	$x^{@str}$	$E\{x \mapsto q\}$	S	f	$var1@_1$
$\Rightarrow$	H	x	$\{x \mapsto q^{@<str>}\}$	S	f	
	H	$x^{@str}$	$E\{x \mapsto q^{@<strs>}\}$	S	f	$var1@_2$
$\Rightarrow$	H	x	$\{x \mapsto q^{@<str +\!\!+ strs>}\}$	S	f	
	$H[q \mapsto (\lambda\overline{x_i}^n.e, E_1)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^n : S$	f	$app2@(^1)$
$\Rightarrow$	K_1	x	$\{x \mapsto q'^{@<strs>}\}$	S	$f \circ \langle strs, q \mapsto \lambda\overline{p_i}^n.q' \rangle$	
	$H[q \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto q\}$	$\overline{p_i}^k : \#q'^{@<strs>} : S$	f	$var2@_1(^2)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_2)]$	x	$\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs, q \mapsto \neg\overline{p_i}^k \rangle$	
	$H[q \mapsto (\lambda\overline{x_i}^k.\lambda\overline{y_i}^n.e, E_1)]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\overline{p_i}^k : \#q'^{@<strs'>} : S$	f	$var2@_2(^3)$
$\Rightarrow$	$H \cup [q' \mapsto (x\ \overline{x_i}^k, E_3)]$	x	$\{x \mapsto q^{@<strs +\!\!+ strs'>}\}$	$\overline{p_i}^k : S$	$f \circ \langle strs', q \mapsto \neg\overline{p_i}^k \rangle$	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts, E_1) : S$	f	$case2@(^4)$
$\Rightarrow$	H	e_k	$E_1 \cup \{\overline{y_{ki} \mapsto p_i^{@<strs>}}\}$	S	$f \circ \langle strs, q \mapsto C_k\ \overline{p_i} \rangle$	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts.\mathbf{otherwise} -> e, E_1) : S$	f	$case2d@(^5)$
$\Rightarrow$	H	e	E_1	S	$f \circ \langle strs, q \mapsto C_k\ \overline{p_i} \rangle$	
	$H[q \mapsto (C_k\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$(alts.v -> e, E_1) : S$	f	$case2v@(^5)$
$\Rightarrow$	H	e_k	$E_1 \cup \{v \mapsto q^{@<strs>}\}$	S	$f \circ \langle strs, q \mapsto C_k\ \overline{p_i} \rangle$	
	$H[q \mapsto (C\ \overline{x_i}, \{\overline{x_i \mapsto p_i}\})]$	x	$E\{x \mapsto q^{@<strs>}\}$	$\#p : S$	f	$var3@(^6)$
$\Rightarrow$	K_6	x	$\{x \mapsto p\}$	S	$f \circ \langle strs, q \mapsto C\ \overline{p_i} \rangle$	

(^{1}) $K_1 = H \cup [q' \mapsto (e, E_1 \cup \{\overline{x_i \mapsto p_i^{@<strs>}}^n\})]$
(^{2}) $E_2 = \{x \mapsto q, \overline{x_i \mapsto p_i}^k\}$
(^{3}) $E_3 = \{x \mapsto q^{@<strs>}, \overline{x_i \mapsto p_i}^k\}$
(^{4}) e_k se corresponde con la alternativa $\overline{C_j\ \overline{y_{ji}} \to e_j}$ en *alts*
(^{5}) C_k no aparece en *alts*
(^{6}) $K_6 = H \cup [p \mapsto (C\ \overline{x_i}, \{\overline{x_i \mapsto p_i^{@<strs>}}\})]$

Figura 5.5: La máquina abstracta Hood compartiendo las clausuras

explicarlas en detalle, queremos recalcar que no existe ninguna regla donde se produzca la clonación de las clausuras, es decir, la compartición se mantiene en todo momento. Llamaremos a esta versión *Hood-Sharing*.

Debido a que las clausuras no están clonadas, ahora es muy importante entender de forma clara no sólo qué clausura está siendo observada, sino también qué clausura la está observando. Por ejemplo, en la expresión **letrec** $x = y^{@str}\ \ldots$ **in** e, tenemos que y es la clausura que está siendo observada, mientras que x es la clausura que la está observando, es decir, x es el observador de y. En esta situación, incluso aunque debemos recordar que y está siendo observada con la marca de observación *str*, cuando actualicemos un valor, la marca de observación debe ser añadida al puntero correspondiente a la variable x y no al puntero correspondiente a la variable y. Por tanto, para hacer esto debemos asociar la observación a la clausura que está observando, no a la que está siendo observada.

Siguiendo las ideas de los modelos anteriores, $x^{@str}$ significa que la clausura x está siendo observada por otra clausura. De forma similar, $q^{@<strs>}$ significa que la clausura localizada en la posición q del *heap* está siendo observada desde otras clausuras con el conjunto de marcas de observación *strs*. Como consecuencia, la clausura q debe ser almacenada en el *heap* sin ninguna marca de observación.

Las reglas $var2@_1$ y $var2@_2$ son exactamente las mismas que las de la Sección 5.2.1, es decir, cuando la compartición no se preservaba, excepto que ahora no es necesario añadir el conjunto de marcas de observación @<*strs*> a la clausura q' que se añade al *heap*, porque la anotación

@<*strs*> significa que otra clausura está observando q'. Sin embargo, las reglas *var1*@$_1$, *var1*@$_2$ y *app2*@ necesitan una pequeña modificación. La única diferencia es que ahora no debemos clonar ninguna clausura, de esta manera, debemos utilizar exactamente las mismas reglas que en la máquina Hugs-Hood.

Téngase en cuenta que en nuestra nueva máquina abstracta no podemos utilizar ni la regla *observer*@, ni la regla *observerH*@. La regla *observer*@ es incorrecta ya que la clausura p se corresponde con la clausura que está siendo observada, no la clausura que está observándola. En el primer modelo, como la clausura p era una copia de la clausura original, podíamos anotarla sin ningún riesgo, ya que sólo se podía entrar a ella a través de la clausura que la estaba observando. Sin embargo, ahora, p no está clonada, por tanto, puede que se entre a ella desde una clausura que está observándola, así como desde cualquier otra clausura. Debido a esto, debemos añadir más reglas para diferenciar ambas situaciones. Resaltaremos que la regla *observerH*@ también es incorrecta en esta situación, ya que no tiene en cuenta quién es el observador de cada clausura bajo observación.

Pasaremos a describir el comportamiento de las nuevas reglas que son necesarias para sustituir la regla *observer*@. Como debemos tener en cuenta tanto al observador como a la clausura observada, necesitamos seguir la evolución del cómputo transmitiendo las correspondientes marcas de las clausuras bajo observación. Véase que las reglas *case2*@, *case2d*@, *case2v*@ y *var3*@ son exactamente las mismas que en la máquina original STG, pero transmiten de forma adecuada las marcas de observación y generan el efecto lateral necesario relativo a la observación.

Las reglas *case2*@, *case2d*@ y *case2v*@ trabajan con las clausuras que han sido reducidas a un constructor y que están siendo observadas bajo el conjunto de marcas de anotaciones *strs*. Por tanto, se debe generar un efecto lateral para observar el constructor y la máquina debe continuar con la evaluación de la alternativa correspondiente de la expresión **case**. Debemos hacer notar que todos los punteros que aparecen en la aplicación del constructor son también marcados como observables; por tanto, se observarán en el caso de que se reduzcan.

La regla *var3*@ se encarga de la actualización de las clausuras. En este caso, se actualiza p almacenando la observación correspondiente y, por tanto, generando el efecto lateral correspondiente. Posteriormente, la máquina continúa con la evaluación de p.

Finalmente recalcaremos que aunque la nueva máquina no es tan simple como la que se presenta en la Sección 5.2.1, aún así consideramos que sigue teniendo una forma bastante simple y compacta.

Ejemplos de evaluación de Hood-Sharing

Para comprender mejor el comportamiento de esta versión, evaluaremos la misma expresión que hemos visto en los casos anteriores. En este caso las observaciones que se obtienen han de ser las mismas que en Hood, pero la compartición de clausuras se mantendrá. Pasemos, por tanto, a ver las diferencias entre el cómputo presentado en el Ejemplo 5.2 y el cómputo que se producirá en esta máquina.

Ejemplo 5.4 Tanto la expresión Haskell como la expresión transformada son las mismas que en el Ejemplo 5.2. Por este motivo no las presentaremos aquí.

Como ya se vio anteriormente, el primer paso de la máquina se corresponde con la reducción

del **letrec**, que produce el siguiente *heap*:

$$
\left\{
\begin{array}{l}
p_0 \mapsto (Int\ x,\ \{x \mapsto 0\#\}) \\
p_1 \mapsto (Int\ x,\ \{x \mapsto 1\#\}) \\
p_2 \mapsto (Int\ x,\ \{x \mapsto 2\#\}) \\
p_3 \mapsto (Cons\ dos\ list1,\ \left\{\begin{array}{l} dos \mapsto p_2 \\ list1 \mapsto p_4 \end{array}\right\}) \\
p_4 \mapsto (NIL,\ \{\}) \\
p_5 \mapsto (\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad CONS\ y\ ys{-}{>} \\
\qquad\qquad\quad \textbf{case}\ length\ ys\ \textbf{of} \\
\qquad\qquad\qquad sol{-}{>}+\ uno\ sol \\
\qquad\qquad NIL{-}{>}\ cero \\
\qquad\qquad ,\ \left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}) \\
p_6 \mapsto (\lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad CONS\ y\ ys{-}{>} \\
\qquad\qquad\quad \textbf{case}\ sum\ ys\ \textbf{of} \\
\qquad\qquad\qquad sol{-}{>}+\ y\ sol \\
\qquad\qquad NIL{-}{>}\ cero \\
\qquad\qquad ,\ \left\{\begin{array}{l} cero \mapsto p_0 \\ sum \mapsto p_6 \end{array}\right\}) \\
p_7 \mapsto (length^{@length},\ \{length \mapsto p_5\}) \\
p_8 \mapsto (lengthO\ list,\ \left\{\begin{array}{l} lentghO \mapsto p_7 \\ list \mapsto p_3 \end{array}\right\}) \\
p_9 \mapsto (sum\ list,\ \left\{\begin{array}{l} sum \mapsto p_6 \\ list \mapsto p_3 \end{array}\right\})
\end{array}
\right\}
$$

Esto provoca que la expresión de control sea la suma. Su evaluación demanda la reducción de `x1` que, en este caso, provocará las mismas observaciones que en la versión STG-Hood, salvo por el hecho de que ahora no se duplican las clausuras. Veamos, por tanto, cómo se reduce la expresión `lengthO list`:

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h_I	$x1$	$\{x1 \mapsto p_8, x2 \mapsto p_9\}$	S_I	$\{\}$	$var1$
$\Longrightarrow$	$h_I - p_8$	$lengthO\ list$	$\{lengthO \mapsto p_7, list \mapsto p_3\}$	$\#P_8 : S_I$	$\{\}$	$app1$
$\Longrightarrow$	$h_I - p_8$	$lengthO$	$\{lengthO \mapsto p_7, list \mapsto p_3\}$	$p_3 : \#P_8 : S_I$	$\{\}$	$var1$
$\Longrightarrow$	$h_I - p_8 - p_7$	$length^{@length}$	$\{length \mapsto p_5\}$	$\#p_7 : p_3 : \#P_8 : S_I$	$\{\}$	$var1@_1$
$\Longrightarrow$	$h_I - p_8 - p_7$	$length$	$\{length \mapsto p_5^{@<length>}\}$	$\#p_7 : p_3 : \#P_8 : S_I$	$\{\}$	$var2$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$length$	$\{length \mapsto p_5^{@<length>}\}$	$p_3 : \#P_8 : S_I$	$\{\}$	$app2@$
$\Longrightarrow$	$\left\{\begin{array}{l} h_I - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \\ p_{10} \mapsto (\textbf{case}\ xs\ \textbf{of}\ alts_1, \{xs \mapsto p_3^{@<length>}, cero \mapsto p_0, length \mapsto p_5, uno \mapsto p_1\}) \end{array}\right\}$	$length$	$\{length \mapsto p_{10}^{@<length>}\}$	$\#P_8 : S_I$	$\{(\text{length},\ p_5 \mapsto\ \lambda\ p_3.\ p_{10})\}$	$var1$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$\textbf{case}\ xs\ \textbf{of}\ alts_1$	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, xs \mapsto p_3^{@<length>}\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\{\ldots\}$	$case1$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	xs	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, xs \mapsto p_3^{@<length>}\}$	$(alts_1, \{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}) : \#p_{10}^{@<length>} : \#P_8 : S_I$	$\{\ldots\}$	$case2@$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$\textbf{case}\ length\ ys\ \textbf{of}\ alts_{1.1}$	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, y \mapsto p_2^{@<length>}, ys \mapsto p_4^{@<length>}\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\left\{\begin{array}{l}(\text{length},\ p_5 \mapsto\ \lambda\ p_3.\ p_{10}) \\ (\text{length},\ p_3 \mapsto Cons\ p_2\ p_4)\end{array}\right\}$	$case1$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$length\ ys$	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, y \mapsto p_2^{@<length>}, ys \mapsto p_4^{@<length>}\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\{\ \ldots \text{Obs. length} \ldots\ \}$	$\ldots$
$\Longrightarrow^*$			$\ldots$ Reducción de la llamada recursiva $\ldots$			
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$cero$	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\left\{\begin{array}{l}\ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil)\end{array}\right\}$	$case2v$
$\Longrightarrow$	$\left\{\begin{array}{c} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \end{array}\right\}$	$+\ uno\ sol$	$\{cero \mapsto p_0, uno \mapsto p_1, length \mapsto p_5, sol \mapsto p_0\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\{\ldots\}$	$app1$
$\Longrightarrow^*$			$\ldots$ Reducción de la suma $\ldots$			
$\Longrightarrow$	$\left\{\begin{array}{l} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \\ p_{11} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \end{array}\right\}$	sol	$\{sol \mapsto p_{11}, sol\# \mapsto 1\#\}$	$\#p_{10}^{@<length>} : \#P_8 : S_I$	$\{\ldots\}$	$var3$
$\Longrightarrow$	$\left\{\begin{array}{l} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \\ p_{11} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \\ p_{10} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \end{array}\right\}$	sol	$\{sol \mapsto p_{10}^{@<length>}\}$	$\#P_8 : S_I$	$\{\ldots\}$	$var3@$
$\Longrightarrow$	$\left\{\begin{array}{l} h_I - p_8 - p_7 \\ p_7 \mapsto (length, \{length \mapsto p_5^{@<length>}\}) \\ p_{11} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \\ p_{10} \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \\ p_8 \mapsto (Int\ sol\#, \{sol\# \mapsto 1\#\}) \end{array}\right\}$	sol	$\{sol \mapsto p_8\}$	S_I	$\left\{\begin{array}{c}\ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\})\end{array}\right\}$	$\ldots$

Veamos ahora el cómputo de la llamada recursiva `length list1`, que provocará la observación de la lista vacía en el fichero de observaciones, así como modificaciones sobre el heap.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	h_I	*length ys*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	S'_I	$\{ \ldots \text{Obs. length} \ldots \}$	*app1*
$\Longrightarrow$	h_I	*length*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ y \mapsto p_2^{@<length>} \\ ys \mapsto p_4^{@<length>} \end{array}\right\}$	$p_4^{@<length>} : S'_I$	$\{\ldots\}$	*app2*
$\Longrightarrow$	h_I	**case** *xs* **of** $alts_1$	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ xs \mapsto p_4^{@<length>} \end{array}\right\}$	S'_I	$\{\ldots\}$	*case1*
$\Longrightarrow$	h_I	*xs*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \\ xs \mapsto p_4^{@<length>} \end{array}\right\}$	$(alts_1, \left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}) : S'_I$	$\{\ldots\}$	*case2@*
$\Longrightarrow$	h_I	*cero*	$\left\{\begin{array}{l} cero \mapsto p_0 \\ uno \mapsto p_1 \\ length \mapsto p_5 \end{array}\right\}$	S'_I	$\left\{\begin{array}{l} \ldots \text{Obs. length} \ldots \\ (\text{length},\ p_4 \mapsto\ Nil) \end{array}\right\}$	*case2v*

En este caso la reducción del segundo argumento de la suma, la clausura p_9 `(sum list)`, no provoca ningún tipo de observación, ya que la variable `list` hace referencia a la clausura p_3 que no ha sido modificada por las observaciones, es decir, no posee ninguna marca de observación. Además, las clausuras alcanzables desde p_3 no poseen marcas de observación.

Por tanto, en este caso se obtiene el mismo fichero de observaciones que en el Ejemplo 5.2, es decir,

$$\left\{\begin{array}{l} (\text{length},\ p_5 \mapsto\ \lambda\ p_3.\ p_{10}) \\ (\text{length},\ p_3 \mapsto\ Cons\ p_2\ p_4) \\ (\text{length},\ p_4 \mapsto\ Nil) \\ (\text{length},\ p_{10} \mapsto\ \{x \mapsto 1\#\}) \end{array}\right\}$$

que se mostraría al usuario de la siguiente forma:

```
-- length
   { \ (_:[]) -> 1 }
```

□

En este modelo se obtienen las mismas anotaciones que en Hood, pero se gana la compartición de cómputos. Lógicamente, esto es una ventaja significativa en cuestión de eficiencia aunque en un contexto de depuración esto no es muy importante.

5.3. Implementación e intérpretes de cada máquina

Con el fin de comprobar varias propiedades de las máquinas anteriores hemos implementado un simulador. En esta sección describiremos brevemente ciertos detalles interesantes de la implementación del simulador. Como indicamos previamente, hemos realizado la implementación de todas las máquinas presentadas en la secciones anteriores. Dicha implementación ha sido utilizada para desarrollar los ejemplos presentados en este capítulo.

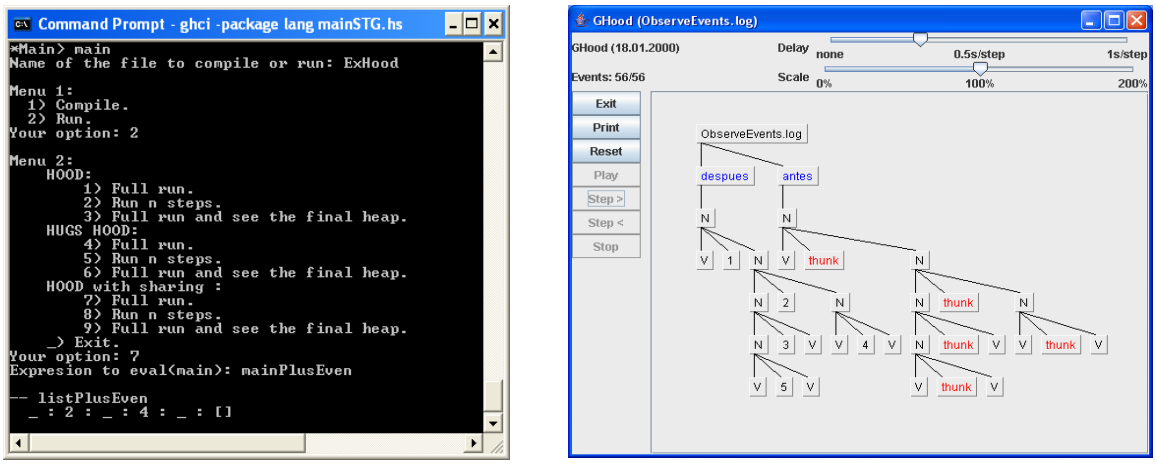


Figura 5.6: Entorno de depuración: Salida en modo texto (izquierda) o salida gráfica (derecha)

Para desarrollar dicha implementación, hemos seguido algunas de las indicaciones de [FW87]. Una de la ventajas de incluir el depurador a nivel de la máquina STG es que se pueden desarrollar diferentes versiones de dicho depurador de forma sencilla, pues sólo es necesario modificar las reglas adecuadas de la máquina STG. En nuestro caso hemos desarrollado un intérprete por cada máquina abstracta presentada aquí, manteniendo el comportamiento común en todas las situaciones posibles. No hemos pretendido en ningún momento desarrollar completamente un compilador, que podía haber sido realizado siguiendo los pasos vistos en [EP03a], sino tener una herramienta que nos permita la reducción de las observaciones teniendo en cuenta las diferentes versiones y así, de esta manera, poder analizar de forma mecánica los cambios en dichas anotaciones.

Esta herramienta nos permite seleccionar en cada situación el tipo de depurador que queremos utilizar (véase la Figura 5.6), es decir, para una situación concreta podemos utilizar un depurador similar al que posee el intérprete Hugs, mientras que en otra situación podemos utilizar un depurador similar a Hood, con compartición de clausuras o sin dicha compartición y en todos los casos se pueden obtener resultados de utilización de *heap*, ver el *heap* final, ver los estados intermedios de las máquinas, ver paso a paso la reducción, etc. Finalmente, queremos comentar que en nuestro entorno de programación no sólo generamos anotaciones planas de observación, sino, que el usuario puede elegir entre obtener la salida a través de un texto plano (mismo estilo que Hood) o a través del uso de GHood [Rei01]. Esto es debido a que durante la ejecución de un cómputo se crean como efecto lateral dos ficheros `hoodObserv.log` y `gHoodObserv.log`. El primero contiene las anotaciones de observación en el formato que se ha presentado en este capítulo (Figura 5.6 a la izquierda) y el segundo contiene la transformación de dichas anotaciones al formato original de Hood, para así podérselas pasar a GHood (Figura 5.6 a la derecha). GHood permite explorar paso a paso la evaluación y ver la secuencia de las reducciones.

La herramienta vista en esta sección ha sido extendida para interpretar la semántica que veremos en los capítulos siguientes.

5.4. Propiedades de estas máquinas

Como se comentó al principio de este capítulo, este trabajo consistía en una aproximación inicial a la depuración a nivel de la máquina *STG*. Para ello tomamos como guía el comportamiento de la librería Hood. Queremos dejar patente que trabajar a nivel de la máquina *STG* en ciertas ocasiones nos da más versatilidad. Sin embargo, debido al hecho de que a este nivel se presentan demasiados detalles, en ciertas ocasiones sólo se consigue enturbiar el comportamiento de las observaciones.

Entre las ventajas de esta aproximación debemos destacar que se han logrado añadir en una misma máquina y con escasas modificaciones varias versiones de depuración y que se ha logrado mantener la compartición de las clausuras que Hood no compartía. Las anotaciones de observación que se producen no son exactamente las mismas que en Hood, pero consideramos que son aún más claras, ya que se produce la anotación completa de qué puntero evaluó a qué expresión, de tal forma que resulta muy sencillo reconstruir la observación completa. No obstante, a cambio se pierde la información relativa a cuándo se comienza a evaluar una clausura.

Sin embargo, hay varias cosas de esta aproximación que son mejorables:

- Las observaciones negativas, del tipo $\neg$ no nos parecen deseables, ni claras para nuestros propósitos. Es más, provocan errores en las observaciones, ya que si en una aplicación una función recibe el mismo valor en varios de sus argumentos y el que provoca su evaluación se corresponde con el que no se encuentra observado, es imposible averiguar si el argumento observado se demandó o no. Estas observaciones son necesarias para el procesamiento del fichero de observaciones y, por tanto, para mostrar al usuario las observaciones adecuadas.

  Afortunadamente, como se comentó en la Sección 5.2.1, estas marcas se pueden eliminar si actualizamos la clausura referente a la aplicación parcial con la λ-abstracción que resulta al realizar dicha aplicación parcial.

- Las anotaciones que se producen no son exactamente las mismas que produce la librería Hood. Aunque, como ya hemos comentado, posiblemente sean incluso mas sencillas. Pero se pierde la información relativa a la demanda de las clausuras.

- La demostración de corrección y equivalencia de estas máquinas sería bastante complicada de llevar a cabo, ya que lo razonable sería realizarla con respecto al código de la librería Hood, que es una librería basada en mónadas.

Debido a todos estos detalles en los capítulos posteriores realizaremos una aproximación formal a la depuración comenzando desde la semántica natural de Sestoft y a partir de ahí desarrollando una máquina abstracta similar a la de Sestoft. Todos estos pasos se realizarán formalmente, de tal modo que no quede ninguna duda de su corrección. Una de las ventajas que posee esta aproximación es que gracias a ella hemos logrado comprender lo sencillo que resulta mantener la compartición de las clausuras respetando el comportamiento original de la librería Hood. Por tanto, una de nuestras pretensiones será mantener, tanto en la semántica como en la máquina abstracta que derivemos, esa compartición de las clausuras, pero sin cambiar para nada el comportamiento del depurador a nivel de las observaciones. Esto creemos que clarificará el comportamiento de la reducción de las expresiones, ya que será similar al del resto de expresiones.

Por otro lado, una de las ventajas de comenzar con la semántica de Sestoft es que las observaciones negativas no se producirán nunca, ya que en Sestoft las funciones se aplican de argumento

en argumento, por tanto, en cada momento observaremos sólo la parte de la aplicación en la que estemos interesados. Aunque perderemos algo de la eficiencia con respecto a la máquina abstracta *STG*, este tipo de modelización será suficiente para nuestros propósitos.

Otra de las conclusiones que podemos sacar de esta primera aproximación, es que aparentemente el tener dos tipos de punteros de punteros diferentes no parece demasiado adecuado. El motivo radica en que se complican bastante las reglas y en todas ellas hemos de tener en cuenta si estamos trabajando con punteros observados o no y cuándo son válidas para ambos tipos de punteros. Por tanto, cuando generemos la máquina abstracta tendremos que buscar algún otro tipo de aproximación a las marcas de observación.

Capítulo 6

Incorporando facilidades de depuración a la semántica natural

En este capítulo extenderemos la semántica natural de Sestoft con características de depuración. La principal ventaja de trabajar a nivel semántico será la claridad. Esto es así porque a este nivel se ocultan muchos detalles de eficiencia y, por tanto, se presenta el comportamiento de forma más sencilla. En ciertas ocasiones, tales como las aplicaciones parciales, el comportamiento que vimos en el capítulo anterior de las observaciones en la máquina *STG* resultaba algo complicado y se mezclaba con las optimizaciones que se suelen desarrollar a nivel de las máquinas abstractas.

Esta extensión de depuración se basa también en la observación de las estructuras. Para desarrollar dichas estructuras nos hemos inspirado nuevamente en el depurador Hood. Esto no implica que hayamos tratado de modelizar fielmente su comportamiento. El principal objetivo ha sido modelar las observaciones de la forma más natural. Además, al habernos inspirado en Hood, esta semántica nos servirá para comprender en detalle el comportamiento interno de Hood, más concretamente esta semántica genera las mismas anotaciones que dicho depurador. Dicha semántica ha sido publicada en [ELR06a].

En el desarrollo de la semántica hemos decidido mantener la compartición de clausuras, que Hood no mantenía. Tenemos que tener en cuenta que trabajar a nivel semántico hace más sencillo la definición del comportamiento de las observaciones, así como la demostración de ciertas propiedades, tales como que las marcas de observación no modifican el comportamiento de la evaluación de las expresiones. Posteriormente utilizaremos este capítulo como base para extender la depuración a la semántica de varios lenguajes paralelos.

Una vez incorporadas en la semántica natural las características de depuración y demostrado que dichas observaciones no provocan un cambio semántico sobre las expresiones observadas, nos inspiraremos en las máquinas *STG* y *Mark-2* para desarrollar una máquina abstracta equivalente a dicha semántica. Tras demostrar la corrección y completitud de dicha máquina con respecto a la semántica habremos comprobado que las marcas de observación en dicha máquina tampoco modifican el resultado final de la evaluación de cualquier expresión.

Tanto en el momento de realizar dicha semántica, como en el momento de derivar dicha máquina, nos hemos basado en el trabajo de Sestoft visto en el Capítulo 2.1.3. El lenguaje que presentaremos se corresponde también con, por un lado, una extensión y, por otro lado, una normalización de dicho lenguaje. El proceso de normalización nos llevará a que tanto las

		-- Expresiones	
e	$\rightarrow$	x	-- variable
	$\mid$	$\lambda\ x.e$	-- abstracción lambda
	$\mid$	$x\ y$	-- aplicación
	$\mid$	**letrec** $\overline{x_i = be_i}$ **in** e	-- let recursivo
	$\mid$	$C\ \overline{x_i}$	-- aplicación de constructores
	$\mid$	**case** x **of** $\overline{C_i\ \overline{x_{ij}} \rightarrow e_i}$	-- expresión case
	$\mid$	$prim$	-- valores primitivos
	$\mid$	$op\ \overline{x_i}$	-- operador primitivo saturado
		-- Ligaduras	
be	$\rightarrow$	e	-- expresiones
	$\mid$	$x^{@str}$	-- **variables observadas**
	$\mid$	$p^{@(r,s)}$	-- **puntero observado (interna)**
	$\mid$	$\lambda^{@\overline{[(r_i,s_i)]}}x.e$	-- **abstracción lambda observada (interna)**
		-- Formas normales débiles de cabeza ($whnf$)	
w	$\rightarrow$	$C\ \overline{x_i}$	-- aplicación de constructores
	$\mid$	$prim$	-- valores primitivos
	$\mid$	$\lambda\ x.e$	-- abstraccion lambda
	$\mid$	$\lambda^{@\overline{[(r_i,s_i)]}}x.e$	-- **abstracción lambda observada (interna)**
		-- Primitivos	
$prim$	$\rightarrow$	$int(1,\ 2,\ \ldots)$	-- enteros primitivos
	$\mid$	$\ldots$	-- otros
		-- Operaciones sobre primitivos	
op	$\rightarrow$	$+$	-- suma
	$\mid$	$\ldots$	-- otras

Figura 6.1: λ-cálculo con depuración

aplicaciones de constructores, como las λ-abstracciones aparezcan únicamente en la parte derecha de las ligaduras de la expresión **letrec**. Esto se debe a que no queremos que las observaciones aparezcan como expresiones normales, sino que aparezcan ligadas en una expresión **letrec**. De esta manera la semántica será más sencilla y las demostraciones de equivalencia se realizarán de forma más comprensible.

6.1. Lenguaje con depuración integrada

Al igual que en el capítulo anterior, pretendemos introducir observaciones de forma similar a las de Hood, es decir, marcaremos de forma especial las estructuras que queremos observar. Por tanto, en nuestro caso debemos ser capaces de anotar cualquier estructura. Tal y como hicimos en la Sección 5.1, crearemos dos ligaduras para cada expresión a observar, una con la expresión

a observar y otra ligadura vinculada a la observación de dicha expresión. Así, la semántica de las observaciones se puede presentar de forma clara y comprensible, ya que están limitados los sitios donde se puede realizar una anotación de observación. El lenguaje que se presenta aquí se corresponde con una pequeña modificación en el lenguaje de Sestoft para incluir una nueva ligadura, como se muestra en la Figura 6.1. En nuestro lenguaje, la expresión $x^{@str}$ es equivalente a la expresión de Hood `observe str x`. Véase que, de acuerdo con la sintaxis del lenguaje, este tipo de expresión sólo puede aparecer como ligadura de un **letrec**.

Además, hemos incorporado directamente los valores primitivos en dicho lenguaje para hacer los ejemplos más comprensibles. De esta manera hemos acercado un poco más dicho lenguaje al lenguaje *STGL*. Como ya se comentó al final de Capítulo 2, estos valores primitivos se considerarán constructores "especiales" de aridad `0`. Así, por un lado, las expresiones **case** trabajan directamente sobre ellos y no es necesario crear expresiones **case** que trabajen específicamente sobre valores primitivos desencapsulados y, por otro lado, no será necesario el trabajo de encapsulamiento y desencapsulamiento. Además, debido a esta consideración, no será necesario incluir ninguna nueva regla que indique que son *whnf*, ya que al considerarse constructores la regla *Cons* original lo indica. Esto no restringe el lenguaje, pues a este nivel se podría equiparar a considerar que no existe ninguna diferencia entre los valores encapsulados y desencapsulados. Aunque en la Figura 6.1 aparecen dichos primitivos, sólo se han añadido por claridad, ya que al considerarlos constructores podríamos considerar que quedan directamente incluidos en la aplicación de constructores.

Nótese que tampoco hemos añadido la alternativa por defecto de las expresiones **case**, ya que dicha alternativa genera demasiadas reglas y puede ser simulada de distintas formas. Recordemos que en el lenguaje STGL-Observado había dos alternativas por defecto. La forma más sencilla de simular la alternativa por defecto **otherwise** consiste en saturar las alternativas de dicha expresión con la expresión asociada a dicha alternativa, es decir, copiar la expresión de dicha alternativa en todos los constructores que no existieran previamente entre las alternativas de la expresión **case** original. Por otro lado, la alternativa por defecto que crea una ligadura se simula con una expresión **letrec**, es decir, **case** e **of** $y \mapsto e'$ se transformaría en **letrec** $y = e$ **in case** y **of otherwise** $\mapsto e'$ y ahora se aplicaría el mismo criterio que en la alternativa por defecto **otherwise**. Fíjese que este proceso de normalización es necesario realizarlo ahora sobre todas las expresiones **case** ya que en el discriminante de dicha expresión sólo puede aparecer una variable. Esto no es relevante en esta semántica, pero será imprescindible cuando tratemos el caso de los lenguajes paralelos. Tampoco se han añadido las expresiones **let** en este lenguaje, ya que como se comentó en la Sección 2.1, a nivel semántico no existe ninguna diferencia entre las expresiones **let** y **letrec**. Sólo a nivel de la máquina abstracta se puede optimizar la evaluación de una expresión **let** con respecto a una expresión **letrec**.

La principal diferencia de este lenguaje con el *STGL* corresponde a las aplicaciones y las λ-abstracciones que en *STGL* llevan varios argumentos y aquí sólo poseen un único argumento. La transformación de las λ-abstracciones de *STGL* a este lenguaje consiste únicamente en añadir paréntesis de forma adecuada, es decir, $\lambda\ x_1\ x_2\ x_3.e$ se transforma en $\lambda\ x_1.(\lambda\ x_2.(\lambda\ x_3.e))$. Mientras que en la aplicación es necesario añadir las ligaduras **letrec** adecuadamente, es decir, si partimos de la expresión $x\ x_1\ x_2\ x_3$ esta se convierte en la expresión **letrec** $y_1 = x\ x_1;\ y_2 = y_1\ x_2$ **in** $y_2\ x_3$ en nuestro lenguaje.

Por otro lado, en nuestro lenguaje aparece un nuevo tipo de forma normal $\lambda^{@[\overline{(r_i,s_i)}]}x.e$: una lambda observada desde diversas posiciones, así como un nuevo tipo de puntero $p^{@(r,s)}$: un puntero

que está siendo observado y hace referencia a la marca de observación correspondiente con su padre. Es importante remarcar que este tipo de expresiones no surgen por medio de ninguna transformación sintáctica y tampoco las puede escribir el programador. Este tipo de anotaciones surge como consecuencia de la reducción de los punteros observados. La anotación (r, s) significa que el padre del puntero es r, en nuestro caso la anotación de la que deriva esta se encuentra en la línea r del fichero, y éste es el s-ésimo hijo de dicho padre. La anotación $[\overline{(r_i, s_i)}]$ significa que la lambda está siendo observada desde varias anotaciones, para cada observación se indica su padre r_i y el número del hijo en el fichero s_i.

Otro detalle importante a resaltar es que las anotaciones en la lambda llevan una lista de posibles observadores, mientras que no existen anotaciones de observación en los constructores y las anotaciones en los punteros llevan una única referencia al observador. Esto es debido a que el comportamiento semántico de los constructores es más simple, ya que en cuanto se reducen se observan y no es necesario volverlos a observar. Sin embargo, las λ-abstracciones se pueden aplicar a distintos argumentos y en cada caso producirán distintos resultados, es decir, cuando se observa una λ-abstracción no sólo se quiere observar la primera aplicación de dicha función, sino cada una de sus aplicaciones. Debido a ello es necesario almacenar en las λ-abstracciones las anotaciones de observaciones, que pueden ser varias. Esta lista nunca puede ser vacía ya que surge cuando la primera observación demanda la evaluación de la lambda, por tanto tiene al menos un elemento. Recalcaremos que los punteros poseen un única marca de observación ya que cuando éstos se evalúan la marca desaparece tras la anotación de la correspondiente observación en el fichero. Pero las λ-abstracciones mantienen las observaciones y una λ-abstracción puede estar observada desde varias marcas, por eso ha de llevar asociada una lista de observaciones.

Por último, comentaremos que este lenguaje es al menos tan expresivo como el lenguaje de Sestoft. Cualquier expresión de Sestoft puede ser transformada a una en nuestro lenguaje únicamente creando las expresiones **letrec** adecuadas. Pero también es tan expresivo como el lenguaje $STGL$, pues sólo es necesario realizar las modificaciones sobre las expresiones explicadas anteriormente.

6.2. Añadiendo la depuración en la semántica natural

Ahora que tenemos el lenguaje de partida, debemos introducir las anotaciones en la semántica de Sestoft. En este caso, al igual que en el capítulo anterior, realizaremos dichas anotaciones en un fichero organizado por líneas. En consecuencia, comenzaremos modificando los juicios de Sestoft añadiendo dicho fichero de observaciones en ellos. Al igual que Sestoft, nosotros mantendremos la propiedad de frescura de forma local a los juicios. De esta manera mantendremos una clara diferencia entre las variables libres y las ligadas. Por tanto, utilizaremos los nombres de variables x e y para variables de programa, es decir, ligadas y utilizaremos p y q como nombres de variables libres, es decir, punteros. Para mantener dicha propiedad de forma local a los juicios será necesario añadir dos conjuntos en la derivación de la semántica, como propusimos en [EP01]. Consecuentemente, nuestros juicios tendrán el siguiente aspecto: $H : e \multimap f \ \Downarrow_{A,C} \ K : w \multimap f'$. Tal y como se ha visto en la semántica de Sestoft, esto significa que la expresión e con sus variables libres ligadas en el *heap* H se evalúa obteniéndose la forma normal w y el nuevo *heap* K. En caso de que en este proceso de cómputo se necesiten variables libres, dichas variables se cogerán del conjunto complementario a $var\ H \cup var\ e \cup A \cup var\ C$, tal y como se indicó en el Capítulo 2.1.3. La diferencia es que ahora se añaden en el fichero f las anotaciones de observaciones que se hayan

$$H : \lambda x.e \looparrowright f \;\Downarrow_{A,C}\; H : \lambda x.e \looparrowright f \qquad \textit{Lam}$$

$$H : C\,\overline{x_i} \looparrowright f \;\Downarrow_{A,C}\; H : C\,\overline{x_i} \looparrowright f \qquad \textit{Cons}$$

$$\frac{H : p \looparrowright f \;\Downarrow_{A,C}\; K : \lambda x.e \looparrowright f' \quad K : e[q/x] \looparrowright f' \;\Downarrow_{A,C}\; L : w \looparrowright f''}{H : p\ q \looparrowright f \;\Downarrow_{A,C}\; L : w \looparrowright f''} \qquad \textit{App}$$

$$\frac{H : e \looparrowright f \;\Downarrow_{A\cup\{p\},C}\; K : w \looparrowright f'}{H \cup [p \mapsto e] : p \looparrowright f \;\Downarrow_{A,C}\; K \cup [p \mapsto w] : w \looparrowright f'} \qquad \textit{Var}$$

$$\frac{H \cup \overline{[p_i \mapsto e_i[\overline{p_i/x_i}]]} : e[\overline{p_i/x_i}] \looparrowright f \;\Downarrow_{A,C}\; K : w \looparrowright f'}{H : \textbf{letrec}\ \overline{x_i = e_i}\ \textbf{in}\ e \looparrowright f \;\Downarrow_{A,C}\; K : w \looparrowright f'} \text{ donde } \overline{p_i} \text{ son frescas} \qquad \textit{Letrec}$$

$$\frac{H : p \looparrowright f \;\Downarrow_{A,C\cup\{\overline{C_i\ \overline{x_{ij}} \to e_i}\}}\; K : C_k\ \overline{q_j} \looparrowright f' \quad K : e_k[\overline{q_j/x_{kj}}] \looparrowright f' \;\Downarrow_{A,C}\; L : w \looparrowright f''}{H : \textbf{case}\ p\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \looparrowright f \;\Downarrow_{A,C}\; L : w \looparrowright f''} \qquad \textit{Case}$$

$$\frac{\overline{H : x_i \looparrowright f_i \;\Downarrow_{A,C}\; K : n_i \looparrowright f_{i+1}}^{\,l}}{H : op\ \overline{x_i}^{\,l} \looparrowright f_1 \;\Downarrow_{A,C}\; L : op\ \overline{n_i}^{\,l} \looparrowright f_{l+1}} \qquad \textit{OpP}$$

Figura 6.2: Reglas de Sestoft adaptadas a las observaciones

producido a lo largo de ese cómputo, modificando por tanto dicho fichero y creando el fichero f'. Al igual que en el capítulo previo, las anotaciones se realizan secuencialmente en dicho fichero. Recordemos que utilizamos la notación $f \circ \langle ann \rangle$ para indicar que hemos añadido la anotación *ann* en una nueva línea al final del fichero f.

Como se observó en la Sección 3.4.3, el tipo de las anotaciones que produce Hood sigue el siguiente patrón: (*portId*, *parent*, *change*), donde *parent*, a su vez, se corresponde con un par (*observeParent*, *observePort*). Nuestras anotaciones van a ser un poco diferentes, debido a que éstas se producen directamente en un fichero y en cada línea del fichero sólo habrá una única anotación. Así pues, el *portId* será sustituido por el número de la línea en el que se realiza la anotación. De esta manera no será necesario añadir dicho campo en la anotación. Por otro lado, debido a que una misma λ-abstracción puede ser observada desde distintas clausuras y nuestra pretensión es mostrar la semántica lo más simple posible, las anotaciones referentes a funciones serán algo diferentes a las originales de Hood. Así, su patrón será el siguiente: $\langle [\overline{(observeParent_i\ observePort_i)}]\ Fun \rangle$. Sin embargo, el resto de anotaciones tendrá el siguiente aspecto: $\langle observeParent\ observePort\ change \rangle$. A partir de este momento, consideraremos que el *portId* será un entero que hará referencia a la línea del fichero donde se ha realizado la anotación correspondiente. Como consecuencia, tenemos que tanto el *observeParent* como el *observePort* también serán enteros. Por otro lado y para manejar correctamente esto, necesitaremos la función *length* f que nos devolverá el número total de líneas del fichero f. Consideraremos que el primer elemento del fichero se encuentra situado en la línea 0. En resumen, las anotaciones serán de la

$$\frac{H : p^{@(length\ \ f,0)} \looparrowright f \circ \langle 0\,0\ Observe\ str\rangle \quad \Downarrow_{A,C} \quad K : w \looparrowright f'}{H : p^{@str} \looparrowright f \quad \Downarrow_{A,C} \quad K : w \looparrowright f'} \; Var@S$$

$$\frac{H : p \looparrowright f \circ \langle r\ s\ Enter\rangle \quad \Downarrow_{A,C} \quad K : C\ \overline{p_i}^k \looparrowright f'}{H : p^{@(r,s)} \looparrowright f \quad \Downarrow_{A,C} \quad K \cup [\overline{q_i \mapsto p_i^{@(length\ \ f',i)}}^k] : C\ \overline{q_i}^k \looparrowright f' \circ \langle r\ s\ Cons\ k\ C\rangle} \; \overline{q_i}\text{ frescas} \; Var@C$$

$$\frac{H : p \looparrowright f \circ \langle r\ s\ Enter\rangle \quad \Downarrow_{A,C} \quad K : \lambda x.e \looparrowright f'}{H : p^{@(r,s)} \looparrowright f \quad \Downarrow_{A,C} \quad K : \lambda^{@[(r,s)]} x.e \looparrowright f'} \; Var@F$$

$$\frac{H : p \looparrowright f \circ \langle r\ s\ Enter\rangle \quad \Downarrow_{A,C} \quad K : \lambda^{@obs} x.e \looparrowright f'}{H : p^{@(r,s)} \looparrowright f \quad \Downarrow_{A,C} \quad K : \lambda^{@(r,s):obs} x.e \looparrowright f'} \; Var@FO$$

$$H : \lambda^{@[\overline{(r_i,s_i)}]} x.e \looparrowright f \quad \Downarrow_{A,C} \quad H : \lambda^{@[\overline{(r_i,s_i)}]} x.e \looparrowright f \; Lam@$$

$$\frac{\begin{array}{c} H : l \looparrowright f \quad \Downarrow_{A,C} \quad K : \lambda^{@[\overline{(r_i,s_i)}]} x.e \looparrowright f' \\ K \cup \left[\begin{array}{l} q \mapsto e[q'/x], \\ q' \mapsto p^{@(length\ \ f',0)} \end{array} \right] : q^{@(length\ \ f',1)} \looparrowright f' \circ \langle [\overline{(r_i\ \ s_i)}]\ Fun\rangle \quad \Downarrow_{A,C} \quad L : w \looparrowright f'' \\ \text{donde } q,\ q' \text{ son frescas} \end{array}}{H : l\ p \looparrowright f \quad \Downarrow_{A,C} \quad L : w \looparrowright f''} \; App@$$

Figura 6.3: Semántica natural de Hood

forma:

$$\begin{array}{rcl} ann & \rightarrow & (observeParent\ observePort)\ Observe\ str \\ & | & (observeParent\ observePort)\ Enter \\ & | & (observeParent\ observePort)\ Cons\ arity\ nameConstr \\ & | & [\overline{(observeParent_i\ observePort_i)}]\ Fun \end{array}$$

Las reglas originales de Sestoft (Figura 2.3) no manejan las observaciones, por tanto, es necesario modificarlas para que añadan el fichero de observaciones. Las reglas nuevas se presentan en la Figura 6.2. Como puede observarse, la evaluación de las expresiones es exactamente la misma que en la semántica de Sestoft. Las únicas diferencias se corresponden con el fichero, que en la derivación de un juicio puede ser modificado y, por tanto, al obtener el resultado de la evaluación de una cierta expresión $H : e \looparrowright f$ no sólo se obtiene el nuevo *heap* y la forma normal, sino que ahora también se obtiene el fichero modificado f'. Además, ahora se han añadido los valores primitivos en dicha semántica. La regla *OpP* modela la semántica de evaluación de un operador primitivo cualquiera de aridad l. Obsérvese que ninguna de estas reglas añade nada al fichero. Esto quiere decir que si partimos de una expresión en la que no tenemos anotada ninguna expresión como observable, entonces el fichero no sufrirá ninguna modificación. Las reglas que van a realizar alguna modificación sobre el fichero son las que trabajan con las nuevas expresiones. Las reglas necesarias para trabajar con las nuevas observaciones se presentan en la Figura 6.3. El sistema

completo está formado por ambos conjuntos de reglas.

Pasaremos ahora a explicar en detalle del comportamiento de las reglas que trabajan con las observaciones:

- Regla $Var@S$. Cuando tenemos que evaluar una variable anotada con la observación *str*, debemos generar una anotación en el fichero de observaciones indicando que hemos empezado a evaluar dicha observación. Esto se realiza añadiendo al fichero está anotación: $\langle 0\,0\ Observe\ str\rangle$. A partir de este momento, tenemos que seguir evaluando dicha clausura pero ya no será necesario recordar el string con el que estaba siendo observada, sino que sólo será necesario recordar dónde se encuentra dicha marca en el fichero (quién es su padre). Por tanto, nos disponemos a reducir la variable con la nueva marca de observación $p^{@(length\ f,0)}$ ($length\ f$ es el número de líneas del fichero f).

- Regla $Var@C$. Cuando tenemos que evaluar una variable con una marca de observación del estilo $p^{@(r,s)}$ se crea una nueva anotación en el fichero de observación indicando que hemos entrado a evaluar dicha clausura. Esta anotación es de la forma $\langle r\ s\ Enter\rangle$. A partir de ese momento, lo único que hay que hacer es reducir la variable p hasta que esta alcance una forma normal. En la evaluación de dicha variable p puede que aparezcan otras anotaciones de observación y como consecuencia el fichero puede ser modificado. La forma normal que se alcanza en esta regla se corresponde con un constructor, lo cual significa que la variable p también ha sido reducida al constructor C_k (recuérdese que si $k = 0$ puede ser un valor primitivo). Por tanto, $p^{@(r,s)}$ ha sido reducida a dicho constructor. Es, en este momento, cuando añadimos una nueva observación al fichero de observaciones indicando que la clausura p cuyo padre de observación se corresponde con (r, s) ha reducido al constructor C_k. La observación que generamos es la siguiente: $\langle r\ s\ Cons\ k\ C\rangle$.

  A partir de este momento hay que crear nuevas clausuras para cada uno de los argumentos de dicho constructor, indicando que éstos están siendo observados, de tal manera que si en algún cómputo se entra a evaluar dichos argumentos se realice la observación pertinente. El padre de observación de dichas clausuras se corresponde con la anotación que acaba de ser realizada. Por tanto, las marcas de observación con las que tenemos que anotar dichas clausuras son del tipo $(length\ f, i)$, donde i se corresponde con el número de argumento del constructor (el número de hijo). Obsérvese que la creación de estas clausuras mantiene la compartición de las clausuras originales $\overline{p_i}$. De esta forma no es necesario duplicar cálculos.

- Regla $Var@F$ y $Var@FO$. Este caso es similar al anterior, tenemos que evaluar $p^{@(r,s)}$. Como en el caso anterior, generamos una anotación en el fichero indicando que hemos entrado a evaluar dicha clausura: $\langle r\ s\ Enter\rangle$. La diferencia surge en que ahora la clausura p reduce a una función, por lo que cada vez que se aplique esta función tendremos que tener en cuenta que está siendo observada. Por este motivo debemos anotarla con la marca de observación. Por otro lado, esta función puede que esté previamente anotada como observable (regla $Var@FO$). En ambos casos tenemos que añadir la nueva marca de observación a la λ-abstracción. Por tanto, es necesario almacenar en las funciones un conjunto de observaciones. La nueva marca de observación a añadir indica que el padre de la observación se corresponde con (r, s). A partir de este momento debemos continuar con la evaluación de la nueva forma normal $\lambda^{@[(r,s)]}x.e$ (regla $Var@F$) o $\lambda^{@(r,s):obs}x.e$ (regla $Var@FO$), una lambda observada, por lo que, deberemos dar una semántica a la reducción de este tipo de nuevas formas normales.

- La regla *Lam*@ indica que $\lambda^{@[\overline{(r_i,s_i)}]}x.e$ es un nuevo tipo de forma normal.

- La regla *App*@ es la regla principal de observación de las aplicaciones y por tanto una de las más complejas. La observación de la λ-abstracción aplicada a una variable conlleva varias tareas. Por un lado, tenemos que tener en cuenta que una lambda puede estar siendo observada desde varias marcas de observación. Por tanto, hemos de tratar la lista que almacena dichas marcas y, por otro lado, cuando observamos una lambda debemos observar tanto su argumento como el resultado que se obtiene al finalizar su aplicación. Primero, al igual que en los casos anteriores, generamos una anotación; en este caso, la anotación $\langle[\overline{(r_i\ s_i)}]\ \mathit{Fun}\rangle$ que indica que la clausura ha sido reducida a una lambda y que estaba siendo observada por la lista de observaciones $[\overline{(r_i, s_i)}]$. Ahora es necesario anotar tanto el argumento como el resultado de la aplicación de la lambda. El argumento se anota como observado añadiendo $q' \mapsto p^{@(\mathit{length}\ f', 0)}$ (véase que $\mathit{length}\ f'$ se corresponde con la línea en la que se ha producido la anotación) en el *heap*, que indica que el argumento está siendo observado y que su padre es $(\mathit{length}\ f', 0)$. Con respecto al resultado, también es necesario añadir una nueva ligadura $q \mapsto e'[q'/x]$. Para observarlo, basta con pasar a evaluar dicha clausura observándola, es decir, $q^{@(\mathit{length}\ f', 1)}$. Recordemos que el puerto 0 de una función se corresponde con su argumento y el puerto 1 de una función se corresponde con el resultado de su aplicación.

Recalcamos que en la regla *App*@ la anotación que se produce en el fichero de observaciones no se ajusta exactamente al formato de las que genera Hood. Ahora bien, posteriormente es muy sencillo realizar un post proceso del fichero para separar cada una de las observaciones y generar las mismas anotaciones que en Hood.

Nótese que no es necesario dar semántica a la aplicación de una variable a otra variable observada $p\ q^{@(r,s)}$, debido a que en la sintaxis se han reducido los sitios en los que puede aparecer una variable observada y en las reglas nunca reemplazamos una variable por un puntero observado.

6.2.1. Ejemplos de evaluación de la semántica natural

Antes de pasar a las definiciones de equivalencia y demostraciones de corrección de nuestra semántica, consideramos necesario presentar varios ejemplos que clarifiquen el comportamiento con respecto a las observaciones de las reglas semánticas. Para ello realizaremos varios ejemplos de cómputo de distintas expresiones. Con ellos procuraremos abarcar los casos más significativos que pueden darse en la evaluación de expresiones anotadas con observaciones. Partiremos de los casos más simples e iremos incrementando la dificultad, de tal forma que será fácil seguirlos.

Ejemplo 6.1 El siguiente ejemplo es simple y carece de interés, salvo por el hecho de que posee dos anotaciones de observación sobre la misma clausura. En él pretendemos llevar a cabo la observación de un único número desde dos marcas diferentes de observación. Partimos de la expresión inicial de Haskell:

```
observe "obs2" (observe "obs1" (10::Int))::Int
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
  diez   = 10
  diez0  = diez@{obs1}
  diez00 = diez0@{obs2}
in diez00
```

A continuación pasaremos a mostrar el árbol de reducción semántico completo para este ejemplo. En cada caso sólo mostraremos las modificaciones que se producen sobre el *heap* y el fichero de observaciones. En varias ocasiones, cuando dicho *heap* o dicho fichero no se modifiquen, aparecerán como una variable del tipo H_i o f_i respectivamente. Cuando eliminemos alguna clausura del *heap*, se mostrará el *heap* resultante completo. Con estas consideraciones, el árbol semántico que se produce es el siguiente:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\} : 10 \looparrowright f_2 \;\Downarrow\; \{\} : 10 \looparrowright f_2}\,Cons}{H_2 : p_1 \looparrowright \left\{\begin{matrix}\cdots \\ 2\ 0\ \mathit{Enter}\end{matrix}\right\} \;\Downarrow\; \{p_1 \mapsto 10\} : 10 \looparrowright \left\{\begin{matrix}\cdots \\ 2\ 0\ \mathit{Cons}\ 0\ 10\end{matrix}\right\}}\,Var}{H_2 : p_1^{@(2,0)} \looparrowright \left\{\begin{matrix}\cdots \\ 0\ 0\ \mathit{Observe}\ \text{obs1}\end{matrix}\right\} \;\Downarrow\; H_3 : 10 \looparrowright f_3}\,Var@C}{\{p_1 \mapsto 10\} : p_1^{@\text{obs1}} \looparrowright f_1 \;\Downarrow\; H_3 : 10 \looparrowright f_3}\,Var@S}{H_1 : p_2 \looparrowright \left\{\begin{matrix}\cdots \\ 0\ 0\ \mathit{Enter}\end{matrix}\right\} \;\Downarrow\; \left\{\begin{matrix}\cdots \\ p_2 \mapsto 10\end{matrix}\right\} : 10 \looparrowright f_3}\,Var}{H_1 : p_2^{@(0,0)} \looparrowright \{0\ 0\ \mathit{Observe}\ \text{obs2}\} \;\Downarrow\; H_4 : 10 \looparrowright \left\{\begin{matrix}\cdots \\ 0\ 0\ \mathit{Cons}\ 0\ 10\end{matrix}\right\}}\,Var@C}{\left\{\begin{matrix}p_1 \mapsto 10 \\ p_2 \mapsto p_1^{@\text{obs1}}\end{matrix}\right\} : p_2^{@\text{obs2}} \looparrowright \{\} \;\Downarrow\; H_4 : 10 \looparrowright f_4}\,Var@S}{\left\{\begin{matrix}p_1 \mapsto 10 \\ p_2 \mapsto p_1^{@\text{obs1}} \\ p_3 \mapsto p_2^{@\text{obs2}}\end{matrix}\right\} : p_3 \looparrowright \{\} \;\Downarrow\; \left\{\begin{matrix}\cdots \\ p_3 \mapsto 10\end{matrix}\right\} : 10 \looparrowright f_4}\,Var$$

$$\dfrac{}{\{\} : \begin{array}{l}\textbf{letrec} \\ \quad diez = 10 \\ \quad diezO = diez^{@obs1} \\ \quad diezOO = diezO^{@obs2} \\ \textbf{in}\ diezOO\end{array} \looparrowright \{\} \;\Downarrow\; \left\{\begin{matrix}p_1 \mapsto 10 \\ p_2 \mapsto 10 \\ p_3 \mapsto 10\end{matrix}\right\} : 10 \looparrowright \left\{\begin{matrix}0\ 0\ \mathit{Observe}\ \text{obs2} \\ 0\ 0\ \mathit{Enter} \\ 0\ 0\ \mathit{Observe}\ \text{obs1} \\ 2\ 0\ \mathit{Enter} \\ 2\ 0\ \mathit{Cons}\ 0\ 10 \\ 0\ 0\ \mathit{Cons}\ 0\ 10\end{matrix}\right\}\,Letrec$$

Por tanto, añadiendo los números de las líneas al fichero de observaciones obtenemos el

siguiente fichero:

Línea	Observación
0	0 0 *Observe* obs2
1	0 0 *Enter*
2	0 0 *Observe* obs1
3	2 0 *Enter*
4	2 0 *Cons* 0 10
5	0 0 *Cons* 0 10

En el fichero de observaciones hay más información de la necesaria para nuestros propósitos. Dicho fichero se encuentra ordenado según la reducción de las distintas clausuras observadas. La primera clausura observada que ha sido demandada ha sido la referente a la observación *obs2* (`diezOO`). Lo siguiente que se ha producido ha sido un comienzo de la reducción de dicha clausura. Antes de que ésta alcanzara su forma normal (línea 5 del fichero) se ha demandado la clausura anotada con la observación *obs1* (`diezO`), se ha entrado a reducir dicha clausura y se ha alcanzado su forma normal (línea 4). Obsérvese que para generar el árbol de observaciones a partir de dicho fichero no son necesarias las marcas de observación *Enter*. La generación de los árboles de observación es muy simple: las anotaciones en el fichero del tipo *Observe str* generan un nuevo árbol de observación anotado con el nombre *str*. A partir de ahí se van analizando las marcas de observación, en este caso del tipo *Cons*: la marca de la línea 4 hace referencia a la línea 2, por tanto su padre se encuentra en la línea 2 y es el hijo 1 de dicho padre. Sucede lo mismo para la marca de la línea 5, que hace referencia a la línea 0, por tanto su padre se encuentra en la línea 0 y es el hijo 1 de dicho padre, es decir, se generan dos árboles de observación independientes, que son los que se muestran a continuación:

Las observaciones que se producen son las resultantes del aplanamiento de este árbol, es decir, mostraría el siguiente mensaje de texto:

```
-- obs1
        10
-- obs2
        10
```

□

Como se puede ver, el árbol semántico que se produce en la evaluación de una expresión es bastante grande, incluso aunque la expresión sea simple. Por tanto, en ejemplos más complicados sólo mostraremos la parte del árbol que consideramos más relevante y sólo mostraremos las partes del *heap* que sean significativas. Con respecto al fichero, veremos en él únicamente las anotaciones que se producen.

El ejemplo que presentaremos a continuación se corresponde con la observación de una estructura algo más complicada, una lista de dos elementos sobre la que aplicaremos la función `length`. Consideraremos `CONS` como constructor de las listas y `NIL` como lista vacía.

Ejemplo 6.2 En este ejemplo queremos observar la demanda producida por la función `length` sobre una lista `list` de dos elementos. Para ello anotaremos la lista con la marca de observación *listaObs* y, por tanto, evaluaremos la aplicación de la función `length` sobre la lista observada `listO`. Para comprobar que los elementos de la lista no se evalúan y que sólo las marcas de observación que son demandadas producen anotaciones en el fichero, uno de los elementos de la lista se encuentra observado (`elemO`). Partimos de la expresión inicial de Haskell:

```
length (observe "listaObs" [observe "elemObs" (1::Int), 6])
  where
    length (x:xs) = 1 + length xs
    length []     = 0
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la expresión de partida, e_0, que se muestra a continuación:

```
letrec
  cero  = 0
  uno   = 1
  elemO = uno@{elemObs}
  seis  = 6
  listO = list@{listaObs}
  list  = CONS elemO xs
  xs    = CONS seis ys
  ys    = NIL
  length = \xs. case xs of
                  CONS y ys -> letrec
                                  sol = length ys
                               in + uno sol
                  NIL       -> cero
in length listO
```

La reducción de la expresión **letrec** genera la siguiente configuración inicial:

$$\left\langle \begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p_1^{@elemObs} \\ p_3 \mapsto 6 \\ p_4 \mapsto p_5^{@listaObs} \\ p_5 \mapsto CONS\ p_2\ p_6 \\ p_6 \mapsto CONS\ p_3\ p_7 \\ p_7 \mapsto NIL \\ p_8 \mapsto \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad CONS\ y\ ys{-}{>} \\ \qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad\quad sol = p_8\ ys \\ \qquad\qquad\quad \textbf{in} + \ p_1\ sol \\ \qquad\qquad NIL{-}{>}\ p_0 \end{array} \right\rangle : p_8\ p_4 \rightarrowtail \{\}$$

Las evaluaciones semánticas más interesantes de resaltar, desde nuestro punto de vista, son las referentes a las observaciones, ya que ellas son las que nos dan información de cómo se reducen las observaciones y qué consecuencias provocan. Dichas evaluaciones se presentan a continuación.

Como primera reducción veremos la evaluación inicial de la marca de observación. Dicha evaluación se produce cuando la función `length` demanda la evaluación de su argumento a través de la reducción de la expresión **case**. Esta reducción, que presentamos a continuación, se corresponde con la evaluación de la clausura p_4 (`listO`), por lo que provoca la reducción de la marca de observación y genera las primeras anotaciones en el fichero de observaciones:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_5 \rightarrowtail \begin{Bmatrix}\dots \\ 0\ 0\ \mathit{Enter}\end{Bmatrix} \Downarrow \{\dots\} : \mathit{CONS}\ p_2\ p_6 \rightarrowtail \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_5^{@(0,0)} \rightarrowtail \{0\ 0\ \mathit{Observe}\ \mathrm{listaObs}\} \Downarrow \begin{Bmatrix}\dots \\ p_9 \mapsto p_2^{@(2,1)} \\ p_{10} \mapsto p_6^{@(2,2)}\end{Bmatrix} : \mathit{CONS}\ p_9\ p_{10} \rightarrowtail \begin{Bmatrix}\dots \\ 0\ 0\ \mathit{Cons}\ 2\ \mathit{CONS}\end{Bmatrix}}\ \mathit{Var@C}}{\{\dots\} : p_5^{@\mathrm{listaObs}} \rightarrowtail \{\} \Downarrow \{\dots\} : \mathit{CONS}\ p_9\ p_{10} \rightarrowtail \{\dots\}}\ \mathit{Var@S}}{\begin{Bmatrix}\dots \\ p_4 \mapsto p_5@\mathrm{listaObs} \\ \dots\end{Bmatrix} : p_4 \rightarrowtail \{\} \Downarrow \begin{Bmatrix}\dots \\ p_4 \mapsto \mathit{CONS}\ p_9\ p_{10} \\ \dots\end{Bmatrix} : \mathit{CONS}\ p_9\ p_{10} \rightarrowtail \{\dots\}}\ \mathit{Var}$$

Posteriormente, cuando la segunda llamada de la función `length` se produce sobre la cola de la lista, es decir, sobre p_{10}, la evaluación que se produce es la siguiente:

$$\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_6 \rightarrowtail \begin{Bmatrix}\dots \\ 2\ 2\ \mathit{Enter}\end{Bmatrix} \Downarrow \{\dots\} : \mathit{CONS}\ p_3\ p_7 \rightarrowtail \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_6^{@(2,2)} \rightarrowtail \{\dots\} \Downarrow \begin{Bmatrix}\dots \\ p_{12} \mapsto p_3^{@(5,1)} \\ p_{13} \mapsto p_7^{@(5,2)}\end{Bmatrix} : \mathit{CONS}\ p_{12}\ p_{13} \rightarrowtail \begin{Bmatrix}\dots \\ 2\ 2\ \mathit{Cons}\ 2\ \mathit{CONS}\end{Bmatrix}}\ \mathit{Var@C}$$

Finalmente, la evaluación de la función `length` se produce sobre la lista correspondiente con la clausura p_{13} y produce la siguiente reducción:

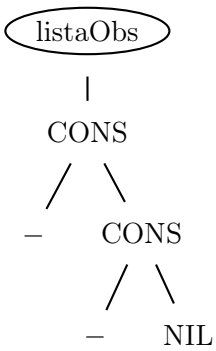
$$\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_7 \rightarrowtail \begin{Bmatrix}\dots \\ 5\ 2\ \mathit{Enter}\end{Bmatrix} \Downarrow \{\dots\} : \mathit{NIL} \rightarrowtail \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_7^{@(5,2)} \rightarrowtail \{\dots\} \Downarrow \{\dots\} : \mathit{CONS}\ p_{12}\ p_{13} \rightarrowtail \begin{Bmatrix}\dots \\ 5\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\end{Bmatrix}}\ \mathit{Var@C}$$

Por tanto, la evaluación de esta expresión produciría la siguiente configuración final:

$$\left\{\begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p_1^{@\text{elemObs}} \\ p_3 \mapsto 6 \\ p_4 \mapsto \ \mathit{CONS}\ p_9\ p_{10} \\ p_5 \mapsto \ \mathit{CONS}\ p_2\ p_6 \\ p_6 \mapsto \ \mathit{CONS}\ p_3\ p_7 \\ p_7 \mapsto \mathit{NIL} \\ p_8 \mapsto \lambda\, xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad \mathit{CONS}\ y1\ ys{-}{>} \\ \qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad sol = p_8\ ys \\ \qquad\qquad\quad \textbf{in}\ +\ p_1\ sol \\ \qquad\qquad \mathit{NIL}{-}{>} p_0 \\ p_9 \mapsto p_2^{@(2,1)} \\ p_{10} \mapsto \mathit{CONS}\ p_{12}\ p_{13} \\ p_{11} \mapsto 1 \\ p_{12} \mapsto p_3^{@(5,1)} \\ p_{13} \mapsto \mathit{NIL} \\ p_{14} \mapsto 0 \end{array}\right\} : 2 \looparrowright \left\{\begin{array}{ll} \text{Línea} & \text{Observación} \\ 0 & 0\ 0\ \mathit{Observe}\ \text{listaObs} \\ 1 & 0\ 0\ \mathit{Enter} \\ 2 & 0\ 0\ \mathit{Cons}\ 2\ \mathit{CONS} \\ 3 & 2\ 2\ \mathit{Enter} \\ 4 & 2\ 2\ \mathit{Cons}\ 2\ \mathit{CONS} \\ 5 & 5\ 2\ \mathit{Enter} \\ 6 & 5\ 2\ \mathit{NIL} \end{array}\right\}$$

Las clausuras p_{11} y p_{14} se corresponden con la clausura que se produce en la función `length` para almacenar el resultado del cómputo parcial de la longitud de la lista a la que le hemos quitado la cabeza.

Analizando el fichero de igual forma que en el ejemplo anterior, se genera el siguiente árbol, donde las marcas _ significan que dicha clausura no ha sido observada, por tanto, no han sido demandadas para la reducción del cómputo:



Al aplanar dicho árbol la observación que se produce es la siguiente:

```
-- listaObs
   _:_:[]
```

En este ejemplo se puede observar que la clausura p_2 no ha sido necesaria para obtener el resultado final del cómputo y, por tanto, no ha sido evaluada, debido a lo cual no se ha producido ninguna marca de observación referente a la evaluación de dicha clausura.

□

Una de las cosas importantes que se muestran en el ejemplo anterior es que las observaciones provocan un aumento de las clausuras en el *heap*. Pero lo más importante que debemos resaltar es que la evaluación de una clausura que se encuentra observando otra (ej. p_4) provoca la evaluación de la clausura que está observando (ej. p_5) y una vez que esta alcanza la forma normal se provocan las observaciones y se actualiza, con dicha forma normal, la clausura que se encontraba observando, es decir, se mantiene el cómputo realizado por la clausura en observación para la actualización de la clausura observante, a diferencia del comportamiento real de Hood.

Los dos ejemplos mostrados anteriormente se corresponden con la evaluación de constructores. En los ejemplos que aparecerán a continuación pasaremos a observar funciones. El primero de ellos se corresponderá con el mismo ejemplo que hemos visto anteriormente, pero ahora en vez de observar la lista, observaremos la función. De esta manera se podrán apreciar claramente las diferencias entre observar un constructor o una función:

Ejemplo 6.3 En este ejemplo queremos observar la función `length`. Para ello la aplicaremos sobre una lista `list` de dos elementos. Por tanto, anotaremos la función con la marca de observación *funObs* y evaluaremos la aplicación de la función `lengthO` sobre la lista `list`. Con ello veremos tanto el resultado de la función como la demanda que dicha función provocará sobre su argumento. Al igual que antes y, para no modificar en gran medida la expresión de partida, mantendremos observado uno de los argumentos de la lista. Partimos de la expresión inicial de Haskell:

```
lengthO [observe "elemObs" (1::Int), 6]
  where
    lengthO = observe "funObs" length

    length (x:xs) = 1 + length xs
    length []     = 0
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la expresión de partida, e_0, que se muestra a continuación:

```
letrec
  cero  = 0
  uno   = 1
  elemO = uno@{elemObs}
  seis  = 6
  list  = CONS elemO xs
  xs    = CONS seis ys
  ys    = NIL
  length = \xs. case xs of
                  CONS y ys -> letrec
                                  sol = length ys
                               in + uno sol
                  NIL       -> cero
  lenghtO = length@{funObs}
in lengthO list
```

En este caso, la reducción de la expresión **letrec** genera la siguiente configuración inicial:

$$\left\{\begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p_1^{@elemObs} \\ p_3 \mapsto 6 \\ p_4 \mapsto CONS\ p_2\ p_5 \\ p_5 \mapsto CONS\ p_3\ p_6 \\ p_6 \mapsto NIL \\ p_7 \mapsto \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad\qquad CONS\ y\ ys{-}{>} \\ \qquad\qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad\qquad\quad sol = p_7\ ys \\ \qquad\qquad\qquad\quad \textbf{in} +\ p_1\ sol \\ \qquad\qquad\qquad NIL{-}{>} p_0 \\ p_8 \mapsto p_7^{@funObs} \end{array}\right\} : p_8\ p_4 \looparrowright \{\}$$

Como primera reducción veremos la evaluación inicial de la marca de observación. Dicha evaluación se produce cuando se va a reducir la clausura referente a la función `lengthO`. Esta reducción, que presentamos a continuación, se corresponde con la evaluación de la clausura p_8 (`lengthO`), por lo que provoca la reducción de la marca de observación y genera las primeras anotaciones en el fichero de observaciones:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_7 \looparrowright \left\{\begin{array}{l}\dots \\ 0\ 0\ Enter\end{array}\right\} \Downarrow \{\dots\} : \lambda\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ Var}{\{\dots\} : p_7^{@(0,0)} \looparrowright \{0\ 0\ Observe\ \text{funObs}\} \Downarrow \{\dots\} : \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ Var@F}{\{\dots\} : p_7^{@\text{funObs}} \looparrowright \{\} \Downarrow \{\dots\} : \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ Var@S}{\left\{\begin{array}{l}\dots \\ p_8 \mapsto p_7@\text{funObs} \\ \dots\end{array}\right\} : p_8 \looparrowright \{\} \Downarrow \left\{\begin{array}{l}\dots \\ p_8 \mapsto \lambda^{[(0,0)]}\, xs.\dots \\ \dots\end{array}\right\} : \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ Var$$

Ahora se provocaría la actualización de la clausura p_8 y se pasaría a evaluar la aplicación de esta función observada sobre la lista `list` (p_4) pero observada (es decir, p_9). Dicha reducción se muestra a continuación:

$$\dfrac{\dots\ p_8\ \dots \quad \left\{\begin{array}{l}\dots \\ p_9 \mapsto p_4^{@(2,0)} \\ p_{10} \mapsto \textbf{case}\ p_9\ \textbf{of}\ \dots\end{array}\right\} : p_{10}^{@(2,1)} \looparrowright \left\{\begin{array}{l}\dots \\ [(0,0)]\ Fun\end{array}\right\} \Downarrow \{\dots\} : 2 \looparrowright \left\{\begin{array}{l}\dots \\ 2\ 1\ Cons\ 0\ 2\end{array}\right\}}{\{\dots\} : p_8\ p_4 \looparrowright \{\dots\} \Downarrow \{\dots\} : 2 \looparrowright \{\dots\}}\ App@$$

$$\dfrac{\dots}{\{\dots\} : p_{10} \looparrowright \left\{\begin{array}{l}\dots \\ 2\ 1\ Enter\end{array}\right\} \Downarrow \{\dots\} : 2 \looparrowright f_1}\ Var \qquad Var@C$$

La reducción de la clausura p_8 no se muestra ya que se corresponde con la reducción vista anteriormente. Por otro lado, la reducción de la clausura p_{10} demanda la evaluación del argumento (p_9) de la función a través de la reducción de la expresión **case** y provoca varias anotaciones en el fichero de observación generando el fichero f_1. Veamos a continuación cómo se produce dicho fichero. Para ello, primero veremos la evaluación de la clausura p_9 que es demandada por la reducción del **case** (véase que la reducción de dicha clausura se corresponde exactamente con la reducción de una lista observada y, por tanto, se generan las mismas observaciones que en el ejemplo anterior):

$$\dfrac{\dfrac{\cdots}{\{\ldots\} : p_4 \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 2\ 0\ \mathit{Enter}\end{array}\right\} \Downarrow \{\ldots\} : \mathit{CONS}\ p_2\ p_5 \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\}}\ \mathit{Var}}{\{\ldots\} : p_4^{@(2,0)} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\} \Downarrow \left\{\begin{array}{l}\cdots\\ p_{11} \mapsto p_2^{@(5,1)}\\ p_{12} \mapsto p_5^{@(5,2)}\end{array}\right\} : \mathit{CONS}\ p_{11}\ p_{12} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 2\ 0\ \mathit{Cons}\ 2\ \mathit{CONS}\end{array}\right\}}\ \mathit{Var@C}$$

A partir de este momento se produce la reducción de la expresión **case**. Dicha reducción demanda la evaluación de la aplicación de la función `length` a la lista de la que se ha eliminado el primer elemento (es decir, p_{12}). La reducción de esta lista produce las siguientes observaciones:

$$\dfrac{\dfrac{\cdots}{\{\ldots\} : p_5 \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 5\ 2\ \mathit{Enter}\end{array}\right\} \Downarrow \{\ldots\} : \mathit{CONS}\ p_3\ p_6 \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\}}\ \mathit{Var}}{\{\ldots\} : p_5^{@(5,2)} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\} \Downarrow \left\{\begin{array}{l}\cdots\\ p_{14} \mapsto p_3^{@(7,1)}\\ p_{15} \mapsto p_6^{@(7,2)}\end{array}\right\} : \mathit{CONS}\ p_{14}\ p_{15} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 5\ 2\ \mathit{Cons}\ 2\ \mathit{CONS}\end{array}\right\}}\ \mathit{Var@C}$$

Finalmente, la evaluación de la función `length` se produce sobre la lista correspondiente a la clausura p_{15} y produce la siguiente reducción, devolviendo el fichero f_1:

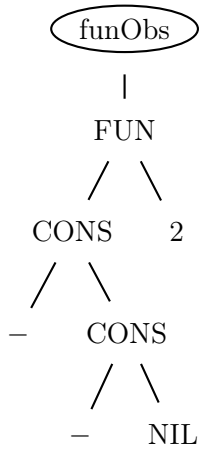
$$\dfrac{\dfrac{\cdots}{\{\ldots\} : p_6 \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 7\ 2\ \mathit{Enter}\end{array}\right\} \Downarrow \{\ldots\} : \mathit{NIL} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\}}\ \mathit{Var}}{\{\ldots\} : p_6^{@(7,2)} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \{\ldots\} \Downarrow \{\ldots\} : \mathit{CONS}\ p_{12}\ p_{13} \mathrel{\rightarrowtail\!\!\!\!\!\!\circ} \left\{\begin{array}{l}\cdots\\ 7\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\end{array}\right\}}\ \mathit{Var@C}$$

Por tanto, la evaluación de esta expresión producirá la siguiente configuración final:

$$\left\{\begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p1^{@\text{elemObs}} \\ p_3 \mapsto 6 \\ p_4 \mapsto \ CONS\ p_2\ p_5 \\ p_5 \mapsto \ CONS\ p_3\ p_6 \\ p_6 \mapsto NIL \\ p_7 \mapsto \lambda\, xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad CONS\ y1\ ys{-}{>} \\ \qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad sol = p_8\ ys \\ \qquad\qquad\quad \textbf{in}\ +\ p_1\ sol \\ \qquad\qquad NIL{-}{>} p_0 \\ p_8 \mapsto \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \ldots \\ p_9 \mapsto CONS\ p_{11}\ p_{12} \\ p_{10} \mapsto 2 \\ p_{11} \mapsto p_2^{@(5,1)} \\ p_{12} \mapsto CONS\ p_{16}\ p_{18} \\ p_{13} \mapsto 1 \\ p_{14} \mapsto CONS\ p_{18}\ p_{19} \\ p_{15} \mapsto 1 \\ p_{14} \mapsto p_3^{@(7,1)} \\ p_{15} \mapsto NIL \\ p_{16} \mapsto 0 \end{array}\right\} : 2 \looparrowright \left\{\begin{array}{rl} \text{Línea} & \text{Observación} \\ 0 & 0\ 0\ \textit{Observe}\ \text{funObs} \\ 1 & 0\ 0\ \textit{Enter} \\ 2 & [(0,0)]\ \textit{Fun} \\ 3 & 2\ 1\ \textit{Enter} \\ 4 & 2\ 0\ \textit{Enter} \\ 5 & 2\ 0\ \textit{Cons}\ 2\ \textit{CONS} \\ 6 & 5\ 2\ \textit{Enter} \\ 7 & 5\ 2\ \textit{Cons}\ 2\ \textit{CONS} \\ 8 & 7\ 2\ \textit{Enter} \\ 9 & 7\ 2\ \textit{Cons}\ 0\ \textit{NIL} \\ 10 & 2\ 1\ \textit{Cons}\ 0\ 2 \end{array}\right\}$$

Las clausuras p_{13} y p_{16} se corresponden con la clausura que se produce en la función `length` para almacenar el resultado del cómputo parcial de la longitud de la lista a la que le hemos quitado la cabeza.

Analizando el fichero de igual forma que en los ejemplos anteriores, con la salvedad de que el hijo 0 de la marca de observación *Fun* se corresponde con el argumento y el hijo 1 se corresponde con el resultado de la función, se genera el siguiente árbol, donde las marcas _ significan que dichas clausuras no han sido observadas, por tanto, no han sido demandadas para la reducción del cómputo:



El aplanamiento de dicho árbol se corresponde con la siguiente observación:

```
-- funObs
   \ (_:_:[]) -> 2
```

En este ejemplo se puede observar que la clausura p_2 no ha sido necesaria para obtener el resultado final del cómputo y, por tanto, no ha sido evaluada, debido a lo cual no se ha producido ninguna marca de observación referente a la evaluación de dicha clausura. □

A continuación se analizará un ejemplo sobre la reducción de una función observada que se aplica varias veces. Como ya estamos habituados a la aplicación de la función `length`, en los siguientes ejemplos seguiremos utilizando dicha función de tal forma que podremos prescindir de ciertos detalles en algunas situaciones.

Ejemplo 6.4 En este ejemplo queremos observar cada una de las aplicaciones de la función `length`, es decir, las aplicaciones que surgen en su definición debido a la llamada recursiva. Para ello, anotaremos la función con la marca de observación *funObs* y evaluaremos la aplicación de la función `length0` sobre la lista de dos elementos `list`. Con ello veremos todas las aplicaciones de esa función sobre las listas parciales que se van creando. Mantendremos también uno de los argumentos de la lista observado. Partimos de la expresión inicial de Haskell:

```
length0 [observe "elem0bs" (1::Int), 6]
  where
    length0 = observe "fun0bs" length

    length (x:xs) = 1 + length0 xs
    length []     = 0
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la expresión de partida, e_0, que se muestra a continuación:

```
letrec
  cero  = 0
  uno   = 1
  elem0 = uno@{elem0bs}
  seis  = 6
  list  = CONS elem0 xs
  xs    = CONS seis ys
```

```
  ys     = NIL
  length = \xs. case xs of
                  CONS y ys -> letrec
                                  sol = lengthO ys
                               in + uno sol
                  NIL       -> cero
  lenghtO = length@{funObs}
in lengthO list
```

Es este caso, la reducción de la expresión **letrec** genera la siguiente configuración inicial:

$$\left\{\begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p_1^{@elemObs} \\ p_3 \mapsto 6 \\ p_4 \mapsto CONS\ p_2\ p_5 \\ p_5 \mapsto CONS\ p_3\ p_6 \\ p_6 \mapsto NIL \\ p_7 \mapsto \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad CONS\ y\ ys -> \\ \qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad\quad sol = p_8\ ys \\ \qquad\qquad\quad \textbf{in} + \ p_1\ sol \\ \qquad\qquad NIL -> p_0 \\ p_8 \mapsto p_7^{@funObs} \end{array}\right\} : p_8\ p_4 \multimap \{\}$$

En este caso, los pasos iniciales de reducción de la clausura p_8 (`lengthO`) son exactamente los mismos que en el ejemplo anterior. La diferencia radica en que a la hora de evaluar la clausura p_{10} producirá distintas anotaciones y por tanto generará un fichero f_1 diferente. Por tanto, nos centraremos en dicha parte. Nos situamos en el momento en que la evaluación de la clausura p_9 habrá dado lugar a la reducción de la lista y la observación de ésta. Es, en este caso, cuando nos pondremos a evaluar por segunda vez la aplicación de la clausura p_8 a la clausura p_{12}, veamos por tanto dicha aplicación:

$$\dfrac{\dots\ p_8\ \dots \quad \left\{\begin{array}{l}\dots \\ p_{14} \mapsto p_{12}^{@(6,0)} \\ p_{15} \mapsto \textbf{case}\ p_{14}\ \textbf{of}\ \dots\end{array}\right\} : p_{15}^{@(6,1)} \multimap \left\{\begin{array}{l}\dots \\ [(0,0)]\ Fun\end{array}\right\} \Downarrow \{\dots\} : 1 \multimap \left\{\begin{array}{l}\dots \\ 6\ 1\ Cons\ 0\ 1\end{array}\right\} \quad App@}{\{\dots\} : p_8\ p_{12} \multimap \{\dots\} \Downarrow \{\dots\} : 1 \multimap \{\dots\}}$$

$$\dfrac{\dots}{\{\dots\} : p_{15} \multimap \left\{\begin{array}{l}\dots \\ 6\ 1\ Enter\end{array}\right\} \Downarrow \{\dots\} : 1 \multimap f_2}\ Var \qquad\qquad Var@C$$

La reducción de la clausura p_8 no se muestra, ya que se corresponde con la reducción de una forma normal. Dicha clausura fue evaluada la primera vez que se demandó y actualizada con su forma normal. Por otro lado, la reducción de la clausura p_{15} demanda la evaluación del argumento (p_{14}) de la función a través de la reducción de la expresión **case** y provoca varias anotaciones en el fichero de observación, generando el fichero f_2. Veamos a continuación cómo se produce dicho fichero. Para ello, primero veremos la evaluación de la clausura p_{14} que es demandada por

la reducción del **case**. En este caso, la clausura p_{14} demandará directamente la evaluación de la clausura p_{12} que a su vez demandará la evaluación de la clausura p_5. Veamos dicho proceso:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_5 \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 5\ 2\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : \mathit{CONS}\ p_3\ p_6 \rightarrow\!\!\!\!\!\rightarrow \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_5^{@(5,2)} \rightarrow\!\!\!\!\!\rightarrow \{\dots\} \Downarrow \left\{\begin{array}{l}\dots\\ p_{16} \mapsto p_3^{@(10,1)}\\ p_{17} \mapsto p_6^{@(10,2)}\end{array}\right\} : \mathit{CONS}\ p_{16}\ p_{17} \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 5\ 2\ \mathit{Cons}\ 2\ \mathit{CONS}\end{array}\right\}}\ \mathit{Var@C}}{\left\{\begin{array}{l}\dots\\ p_{12} \mapsto p_5^{@(5,2)}\end{array}\right\} : p_{12} \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 6\ 0\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : \mathit{CONS}\ p_{16}\ p_{17} \rightarrow\!\!\!\!\!\rightarrow \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_{12}^{@(6,0)} \rightarrow\!\!\!\!\!\rightarrow \{\dots\} \Downarrow \left\{\begin{array}{l}\dots\\ p_{18} \mapsto p_{16}^{@(11,1)}\\ p_{19} \mapsto p_{17}^{@(11,2)}\end{array}\right\} : \mathit{CONS}\ p_{18}\ p_{19} \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 6\ 0\ \mathit{Cons}\ 2\ \mathit{CONS}\end{array}\right\}}\ \mathit{Var@C}$$

Nótese que los argumentos del constructor de la segunda lista están doblemente observados desde las clausuras p_{18} y p_{19} y desde las clausuras p_{16} y p_{17}. Esto es debido a que son argumentos tanto de la primera llamada a la función como de la segunda llamada a la función, y, por tanto, se observan desde dos sitios diferentes. Ya sólo nos faltaría ver la reducción de la evaluación de la función `lengthO` sobre la lista correspondiente a la clausura p_{19}. Como en el caso anterior, empezaremos por ver la reducción de la aplicación:

$$\dfrac{\dots p_8 \dots \quad \dfrac{\dfrac{\dots}{\{\dots\} : p_{22} \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 12\ 1\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : 0 \rightarrow\!\!\!\!\!\rightarrow f_3}\ \mathit{Var}}{\left\{\begin{array}{l}\dots\\ p_{21} \mapsto p_{19}^{@(12,0)}\\ p_{22} \mapsto \mathbf{case}\ p_{21}\ \mathbf{of}\ \dots\end{array}\right\} : p_{22}^{@(12,1)} \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ [(0,0)]\ \mathit{Fun}\end{array}\right\} \Downarrow \{\dots\} : 0 \rightarrow\!\!\!\!\!\rightarrow \left\{\begin{array}{l}\dots\\ 12\ 1\ \mathit{Cons}\ 0\ 0\end{array}\right\}}\ \mathit{Var@C}}{\{\dots\} : p_8\ p_{19} \rightarrow\!\!\!\!\!\rightarrow \{\dots\} \Downarrow \{\dots\} : 0 \rightarrow\!\!\!\!\!\rightarrow \{\dots\}}\ \mathit{App@}$$

Ahora se produce la demanda sobre el cuerpo de la función (p_{22}), que provoca la demanda de

la clausura p_{21} y, por tanto, produce varias anotaciones en el fichero. Veámoslo en detalle:

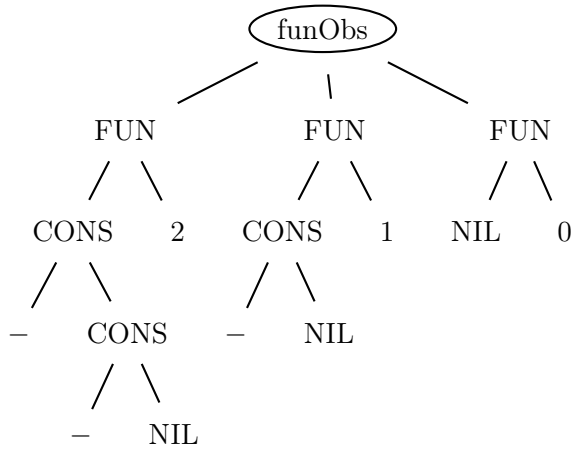
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\left\{\begin{array}{l}\dots\\ p_6 \mapsto \mathit{NIL}\end{array}\right\} : p_6 \rightsquigarrow \left\{\begin{array}{l}\dots\\ 10\ 2\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_6^{@(10,2)} \rightsquigarrow \{\dots\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \left\{\begin{array}{l}\dots\\ 10\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\end{array}\right\}}\ \mathit{Var@C}}{\left\{\begin{array}{l}\dots\\ p_{17} \mapsto p_6^{@(10,2)}\end{array}\right\} : p_{17} \rightsquigarrow \left\{\begin{array}{l}\dots\\ 11\ 2\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_{17}^{@(11,2)} \rightsquigarrow \{\dots\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \left\{\begin{array}{l}\dots\\ 11\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\end{array}\right\}}\ \mathit{Var@C}}{\left\{\begin{array}{l}\dots\\ p_{19} \mapsto p_{17}^{@(11,2)}\end{array}\right\} : p_{19} \rightsquigarrow \left\{\begin{array}{l}\dots\\ 12\ 0\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_{19}^{@(12,0)} \rightsquigarrow \{\dots\} \Downarrow \{\dots\} : \mathit{NIL} \rightsquigarrow \left\{\begin{array}{l}\dots\\ 12\ 0\ \mathit{Cons}\ 0\ \mathit{NIL}\end{array}\right\}}\ \mathit{Var@C}$$

Por tanto, la evaluación de esta expresión producirá la siguiente configuración final:

$$
\left\{
\begin{array}{l}
p_0 \mapsto 0 \\
p_1 \mapsto 1 \\
p_2 \mapsto p1^{@\text{elemObs}} \\
p_3 \mapsto 6 \\
p_4 \mapsto \ \mathit{CONS}\ p_2\ p_5 \\
p_5 \mapsto \ \mathit{CONS}\ p_3\ p_6 \\
p_6 \mapsto \mathit{NIL} \\
p_7 \mapsto \lambda\, xs.\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad \mathit{CONS}\ y1\ ys{-}{>} \\
\qquad\qquad\qquad\quad \textbf{letrec} \\
\qquad\qquad\qquad\qquad sol = p_8\ ys \\
\qquad\qquad\qquad\quad \textbf{in}\ +\ p_1\ sol \\
\qquad\qquad\qquad \mathit{NIL}{-}{>}\ p_0 \\
p_8 \mapsto \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \ldots \\
p_9 \mapsto \mathit{CONS}\ p_{11}\ p_{12} \\
p_{10} \mapsto 2 \\
p_{11} \mapsto p_2^{@(5,1)} \\
p_{12} \mapsto \mathit{CONS}\ p_{16}\ p_{17} \\
p_{13} \mapsto 1 \\
p_{14} \mapsto \mathit{CONS}\ p_{18}\ p_{19} \\
p_{15} \mapsto 1 \\
p_{16} \mapsto p_3^{@(10,1)} \\
p_{17} \mapsto \mathit{NIL} \\
p_{18} \mapsto p_{16}^{@(11,1)} \\
p_{19} \mapsto \mathit{NIL} \\
p_{20} \mapsto 0 \\
p_{21} \mapsto \mathit{NIL} \\
p_{22} \mapsto 0
\end{array}
\right\}
: 2 \looparrowright
\left\{
\begin{array}{rl}
\text{Línea} & \text{Observación} \\
0 & 0\ 0\ \mathit{Observe}\ \text{funObs} \\
1 & 0\ 0\ \mathit{Enter} \\
2 & [(0,0)]\ \mathit{Fun} \\
3 & 2\ 1\ \mathit{Enter} \\
4 & 2\ 0\ \mathit{Enter} \\
5 & 2\ 0\ \mathit{Cons}\ 2\ \mathit{CONS} \\
6 & [(0,0)]\ \mathit{Fun} \\
7 & 6\ 1\ \mathit{Enter} \\
8 & 6\ 0\ \mathit{Enter} \\
9 & 5\ 2\ \mathit{Enter} \\
10 & 5\ 2\ \mathit{Cons}\ 2\ \mathit{CONS} \\
11 & 6\ 0\ \mathit{Cons}\ 2\ \mathit{CONS} \\
12 & [(0,0)]\ \mathit{Fun} \\
13 & 12\ 1\ \mathit{Enter} \\
14 & 12\ 0\ \mathit{Enter} \\
15 & 11\ 2\ \mathit{Enter} \\
16 & 10\ 2\ \mathit{Enter} \\
17 & 10\ 2\ \mathit{Cons}\ 0\ \mathit{NIL} \\
18 & 11\ 2\ \mathit{Cons}\ 0\ \mathit{NIL} \\
19 & 12\ 0\ \mathit{Cons}\ 0\ \mathit{NIL} \\
20 & 12\ 1\ \mathit{Cons}\ 0\ 0 \\
21 & 6\ 1\ \mathit{Cons}\ 0\ 1 \\
22 & 2\ 1\ \mathit{Cons}\ 0\ 2
\end{array}
\right\}
$$

Las clausuras p_{13} y p_{20} se corresponden con la clausura que se produce en la función `length` para almacenar el resultado del cómputo parcial de la longitud de la lista a la que le hemos quitado la cabeza.

Analizando el fichero de igual forma que en los ejemplos anteriores, con la salvedad de que ahora la marca de observación inicial $(0,0)$ tiene varios hijos, se genera el siguiente árbol, donde las marcas _ significan que dichas clausuras no han sido observadas, por tanto, no han sido demandadas para la reducción del cómputo:



El aplanamiento de dicho árbol se corresponde con la siguiente observación:

```
-- funObs
   { \ (_:_:[]) -> 2
   , \ (_:[]) -> 1
   , \ [] -> 0
   }
```

Como se ha podido comprobar, sucede lo mismo que en los ejemplos anteriores, la clausura p_2 no ha sido demandada y, por tanto, no ha sido evaluada ni ha producido ninguna observación □

En el último ejemplo significativo, veremos la evaluación de una función observada desde varias marcas de observación. Dicho ejemplo se basará en el anterior y anotaremos la llamada recursiva que se produce dentro de la función `length` con otra marca de observación.

Ejemplo 6.5 En este ejemplo queremos observar cada una de las aplicaciones de la función `length` pero desde varias marcas de observación. Queremos observar cuándo se realiza la aplicación recursiva y diferenciarla de la aplicación inicial de la función length. Para ello la aplicaremos sobre una lista de dos elementos `list`. Por tanto, anotaremos la función con la marca de observación *funObs* y evaluaremos la aplicación de la función `length0` sobre la lista `list`. Con ello veremos todas las aplicaciones de esa función sobre las listas parciales que se van creando. Mantendremos también uno de los argumentos de la lista observado. Partimos de la expresión inicial de Haskell:

```
length0 (observe "listaObs" [observe "elemObs" (1::Int), 6])
  where
    length00 = observe "funInterna" length0
    length0 = observe "funObs" length

    length (x:xs) = 1 + length00 xs
    length []     = 0
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la expresión de partida, e_0, que se muestra a continuación:

```
letrec
  cero  = 0
```

```
  uno    = 1
  elem0  = uno@{elemObs}
  seis   = 6
  list   = CONS elem0 xs
  xs     = CONS seis ys
  ys     = NIL
  length = \xs. case xs of
                  CONS y ys -> letrec
                                  sol = length00 ys
                               in + uno sol
                  NIL       -> cero
  lenght0 = length@{funObs}
  lenght00 = length0@{funInterna}
in length0 list
```

Es este caso, la reducción de la expresión **letrec** genera la siguiente configuración inicial:

$$\left\{\begin{array}{l} p_0 \mapsto 0 \\ p_1 \mapsto 1 \\ p_2 \mapsto p_1^{@elemObs} \\ p_3 \mapsto 6 \\ p_4 \mapsto \mathit{CONS}\ p_2\ p_5 \\ p_5 \mapsto \mathit{CONS}\ p_3\ p_6 \\ p_6 \mapsto \mathit{NIL} \\ p_7 \mapsto \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad \mathit{CONS}\ y\ ys -> \\ \qquad\qquad\quad \textbf{letrec} \\ \qquad\qquad\qquad sol = p_9\ ys \\ \qquad\qquad\quad \textbf{in} + \ p_1\ sol \\ \qquad\qquad \mathit{NIL} -> p_0 \\ p_8 \mapsto p_7^{@funObs} \\ p_9 \mapsto p_8^{@funInterna} \end{array}\right\} : p_8\ p_4 \looparrowright \{\}$$

Aquí resaltaremos únicamente el computo referente a la primera aplicación de la función `lengthOO` sobre la lista *xs*. Para ello primero veremos la reducción de la clausura `legthOO` (es decir, p_9). Dicho computó se muestra a continuación:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{\dots\} : p_8 \looparrowright \left\{\begin{array}{l}\dots \\ 6\ 0\ \mathit{Enter}\end{array}\right\} \Downarrow \left\{\begin{array}{l}\dots \\ p_8 \mapsto \lambda^{[(0,0)]}\ xs.\dots\end{array}\right\} : \lambda^{[(0,0)]}\ xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ \mathit{Var}}{\{\dots\} : p_8^{@(6,0)} \looparrowright \{0\ 0\ \mathit{Observe}\ \text{funInterna}\} \Downarrow \{\dots\} : \lambda^{[(6,0),(0,0)]}\ xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ \mathit{Var@F}}{\{\dots\} : p_8^{@\text{funInterna}} \looparrowright \{\} \Downarrow \{\dots\} : \lambda^{[(6,0),(0,0)]}\ xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ \mathit{Var@S}}{\left\{\begin{array}{l}\dots \\ p_9 \mapsto p_8@\text{funInterna}\end{array}\right\} : p_9 \looparrowright \{\dots\} \Downarrow \left\{\begin{array}{l}\dots \\ p_9 \mapsto \lambda^{[(0,0)]}\ xs.\dots\end{array}\right\} : \lambda^{[(6,0),(0,0)]}\ xs.\textbf{case}\ xs\ \textbf{of}\ \dots \looparrowright \{\dots\}}\ \mathit{Var}$$

Ya sabemos cómo reduce p_9, veamos por tanto su primera aplicación:

$$\dfrac{\dots\ p_9\ \dots \quad \dfrac{\dots}{\{\dots\} : p_{16} \looparrowright \left\{\begin{array}{l}\dots\\ 8\ 1\ \mathit{Enter}\end{array}\right\} \Downarrow \{\dots\} : 1 \looparrowright f_2}\ \mathit{Var} \quad \dfrac{\{\dots\} : p_{16}^{@(8,1)} \looparrowright \left\{\begin{array}{l}\dots\\ [(6,0),(0,0)]\ \mathit{Fun}\end{array}\right\} \Downarrow \{\dots\} : 1 \looparrowright \left\{\begin{array}{l}\dots\\ 8\ 1\ \mathit{Cons}\ 0\ 1\end{array}\right\}}{\{\dots\} : p_9\ p_{13} \looparrowright \{\dots\} \Downarrow \{\dots\} : 1 \looparrowright \{\dots\}}\ \mathit{Var@C}}{}\ \mathit{App@}$$

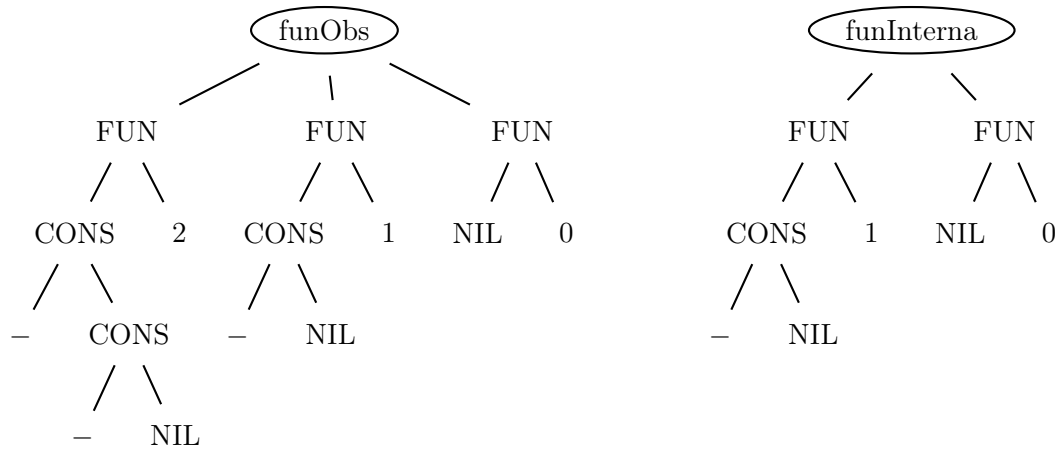
Por tanto, la evaluación de esta expresión producirá la siguiente configuración final:

$$\left\{\begin{array}{l}
p_0 \mapsto 0\\
p_1 \mapsto 1\\
p_2 \mapsto p1^{@\mathrm{elemObs}}\\
p_3 \mapsto 6\\
p_4 \mapsto \ \mathit{CONS}\ p_2\ p_5\\
p_5 \mapsto \ \mathit{CONS}\ p_3\ p_6\\
p_6 \mapsto \mathit{NIL}\\
p_7 \mapsto \lambda\, xs.\ \textbf{case}\ xs\ \textbf{of}\\
\qquad\qquad\qquad \mathit{CONS}\ y1\ ys{-}{>}\\
\qquad\qquad\qquad\quad \textbf{letrec}\\
\qquad\qquad\qquad\qquad sol = p_8\ ys\\
\qquad\qquad\qquad\quad \textbf{in}\ +\ p_1\ sol\\
\qquad\qquad\qquad \mathit{NIL}{-}{>}\ p_0\\
p_8 \mapsto \lambda^{[(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots\\
p_9 \mapsto \lambda^{[(6,0),(0,0)]}\, xs.\textbf{case}\ xs\ \textbf{of}\ \dots\\
p_{10} \mapsto \mathit{CONS}\ p_{12}\ p_{13}\\
p_{11} \mapsto 2\\
p_{12} \mapsto p_2^{@(5,1)}\\
p_{13} \mapsto \mathit{CONS}\ p_{17}\ p_{18}\\
p_{14} \mapsto 1\\
p_{15} \mapsto \mathit{CONS}\ p_{19}\ p_{20}\\
p_{16} \mapsto 1\\
p_{17} \mapsto p_3^{@(12,1)}\\
p_{18} \mapsto \mathit{NIL}\\
p_{19} \mapsto p_{17}^{@(13,1)}\\
p_{20} \mapsto \mathit{NIL}\\
p_{21} \mapsto 0\\
p_{22} \mapsto \mathit{NIL}\\
p_{23} \mapsto 0
\end{array}\right\} : 2 \looparrowright \left\{\begin{array}{rl}
\text{Línea} & \text{Observación}\\
0 & 0\ 0\ \mathit{Observe}\ \mathrm{funObs}\\
1 & 0\ 0\ \mathit{Enter}\\
2 & [(0,0)]\ \mathit{Fun}\\
3 & 2\ 1\ \mathit{Enter}\\
4 & 2\ 0\ \mathit{Enter}\\
5 & 2\ 0\ \mathit{Cons}\ 2\ \mathit{CONS}\\
6 & 0\ 0\ \mathit{Observe}\ \mathrm{funInterna}\\
7 & 6\ 0\ \mathit{Enter}\\
8 & [(6,0),(0,0)]\ \mathit{Fun}\\
9 & 8\ 1\ \mathit{Enter}\\
10 & 8\ 0\ \mathit{Enter}\\
11 & 5\ 2\ \mathit{Enter}\\
12 & 5\ 2\ \mathit{Cons}\ 2\ \mathit{CONS}\\
13 & 8\ 0\ \mathit{Cons}\ 2\ \mathit{CONS}\\
14 & [(6,0),(0,0)]\ \mathit{Fun}\\
15 & 14\ 1\ \mathit{Enter}\\
16 & 14\ 0\ \mathit{Enter}\\
17 & 13\ 2\ \mathit{Enter}\\
18 & 12\ 2\ \mathit{Enter}\\
19 & 12\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\\
20 & 13\ 2\ \mathit{Cons}\ 0\ \mathit{NIL}\\
21 & 14\ 0\ \mathit{Cons}\ 0\ \mathit{NIL}\\
22 & 14\ 1\ \mathit{Cons}\ 0\ 0\\
23 & 8\ 1\ \mathit{Cons}\ 0\ 1\\
24 & 2\ 1\ \mathit{Cons}\ 0\ 2
\end{array}\right\}$$

Las clausuras p_{14} y p_{21} se corresponden con la clausura que se produce en la función `length` para almacenar el resultado del cómputo parcial de la longitud de la lista a la que le hemos quitado la cabeza.

Analizando el fichero de igual forma que en los ejemplos anteriores, con la salvedad de que ahora tenemos una lambda observada desde varios sitios y, por tanto, debemos duplicarla en cada

sitio del árbol donde se observa, se genera el siguiente árbol, donde las marcas _ significan que dichas clausuras no han sido observadas, por tanto, no han sido demandadas para la reducción del cómputo:



El aplanamiento de dicho árbol se corresponde con la siguiente observación:

```
-- funObs
   { \ (_:_:[]) -> 2
   , \ (_:[]) -> 1
   , \ [] -> 0
   }
-- funInterna
   { \ (_:[]) -> 1
   , \ [] -> 0
   }
```

Sucede lo mismo que en los ejemplos anteriores, la clausura p_2 no ha sido demandada y, por tanto, no ha sido evaluada ni ha producido ninguna observación. □

6.2.2. Corrección y equivalencia de la semántica

Ahora que intuitivamente comprendemos el comportamiento de la semántica, pasemos a demostrar formalmente su corrección y completitud.

Al igual que Sestoft, lo primero que debemos comprobar es que en nuestra semántica no se produce ninguna captura de variable. Si hubiera captura de variables la expresión podría no reducir a su valor. La siguiente proposición, que es equivalente a la Proposición 2.1 de Sestoft, demuestra que en cualquier derivación se mantiene una clara diferencia entre las variables libres (es decir, punteros) y ligadas de la expresiones (es decir, de programa). Ya que ambos son conjuntos disjuntos.

Proposición 6.1 *Si* $H : e \Downarrow_{A,C} K : w$ *es una derivación de nuestra semántica,* $fv\ e \cap bv\ e = \varnothing$ *y para toda* $(p \mapsto e') \in \Gamma$, $fv\ e' \cap bv\ e' = \varnothing$, *entonces:*

1. $fv\ w \cap bv\ w = \varnothing$ *y*
2. *Para toda* $(p \mapsto e') \in K$, $fv\ e' \cap bv\ e' = \varnothing$ *y*

3. Las premisas (1) y (2) de la proposición se cumplen para todo juicio intermedio de la derivación.

Demostración 1 La demostración se realiza por inducción sobre la profundidad de la derivación de $\Downarrow_{A,C}$, presentaremos un esquema intuitivo sobre dicha demostración.

Las reglas más interesantes de analizar corresponden con las reglas *Letrec*, *Var*, *App*, *Case*, *App*@ y *Var*@C, pues en todas ellas o se eliminan punteros del *heap*, o se añaden punteros al *heap*, o se sustituyen variables por punteros. La única regla que elimina punteros del *heap* corresponde con la regla *Var* que elimina el puntero bajo evaluación. Esta regla mete el nombre de dicho puntero en el conjunto A para que cuando creemos nuevos punteros no se puedan crear con dicho nombre y, de esta manera, no se produzca captura de variables. Las reglas que añaden punteros son las reglas *Letrec*, *App*@ y *Var*@C, los punteros añadidos por estas reglas son *frescos* con respecto a los conjuntos A y C y al *heap*, por tanto, tampoco provocan ninguna captura de variables. Las reglas que realizan sustituciones de variables por punteros son las reglas *App*, *Case* y *App*@. Las reglas *App* y *App*@ reemplazan la variable x de la expresión e, que ha dejado de ser una variable ligada por la λ, por una variable fresca, consecuentemente, tampoco pueden provocar una captura de variables y además mantienen la distinción entre punteros y variables ligadas. La regla *Case* durante la evaluación del discriminante e almacena las alternativas en el conjunto C para que no se puedan utilizar los nombres de variables como variables frescas. Tras el cómputo del discriminante elimina dichas alternativas de C y sustituye las variables $\overline{x_i}$ por los punteros $\overline{q_i}$; en este caso no puede haber una captura de variables y se mantienen la distinción entre variables y punteros, ya que por un lado por hipótesis de inducción sabemos que $\overline{q_i}$ son punteros y, por tanto mantienen la distinción con las variables, y, por otro lado, las variables $\overline{x_i}$ de la expresión e_k han dejado de estar ligadas por el constructor C_k, luego la sustitución mantiene la diferencia de variables y punteros y no genera captura de variables.

La primera cuestión importante que debemos probar es que las marcas de observación no cambian la semántica de una expresión, es decir, si evaluamos una expresión marcada y la equivalente sin marcas obtenemos las mismas formas normales. Recalcaremos que esta propiedad se debe mantener ya que la depuración no debe modificar la evolución del cómputo de los programas en Haskell.

La principal diferencia de nuestra semántica con respecto a la de Sestoft consiste en que en nuestra semántica las expresiones pueden tener marcas de observación. Por tanto, para comparar ambas semánticas y definir una relación de equivalencia primero deberemos eliminar dichas marcas. La función R que se define a continuación transforma cualquier expresión de nuestro lenguaje, que nosotros llamaremos Sestoft$^{@}$, en una expresión sin observaciones.

Definición 6.1 La función que elimina las observaciones es la siguiente R : Sestoft$^{@}$ $\rightarrow$ Sestoft.

Se define de forma recursiva, todos los casos son triviales excepto las expresiones observadas:

$$
\begin{array}{lcl}
\mathsf{R} \;::\; \text{Sestoft}^{@} \to \text{Sestoft} & & \\
\mathsf{R}\ x & \stackrel{\text{def}}{=} & x \\
\mathsf{R}\ \lambda\ x.e & \stackrel{\text{def}}{=} & \lambda\ x.\mathsf{R}\ e \\
\mathsf{R}\ x\ y & \stackrel{\text{def}}{=} & x\ y \\
\mathsf{R}\ (\textbf{letrec}\ \overline{x_i\ =\ e_i}\ \textbf{in}\ e) & \stackrel{\text{def}}{=} & \textbf{letrec}\ \overline{x_i\ =\ \mathsf{RB}\ e_i}\ \textbf{in}\ \mathsf{R}\ e \\
\mathsf{R}\ (C\ \overline{x_i}) & \stackrel{\text{def}}{=} & C\ \overline{x_i} \\
\mathsf{R}\ (\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i}) & \stackrel{\text{def}}{=} & \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to \mathsf{R}\ e_i} \\
\mathsf{R}\ prim & \stackrel{\text{def}}{=} & prim \\
\mathsf{R}\ op\ \overline{x_i} & \stackrel{\text{def}}{=} & op\ \overline{x_i}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{RB} \;::\; \text{Sestoft}^{@}\text{Ligadura} \to \text{SestoftLigadura} & & \\
\mathsf{RB}\ e & \stackrel{\text{def}}{=} & \mathsf{R}\ e \\
\mathsf{RB}\ x^{@str} & \stackrel{\text{def}}{=} & x \\
\mathsf{RB}\ p^{@(r,s)} & \stackrel{\text{def}}{=} & p \\
\mathsf{RB}\ \lambda^{@\overline{[(r_i,s_i)]}}x.e & \stackrel{\text{def}}{=} & \lambda x.\mathsf{R}\ e
\end{array}
$$

Esta función se extiende de forma natural para trabajar con *heaps* y configuraciones. Básicamente,

- $\mathsf{R}\ H$ se define como $\{p \mapsto \mathsf{RB}\ e \mid (p \mapsto e) \in H\}$ y
- $\mathsf{R}\ (H : e) = \mathsf{R}\ H : \mathsf{R}\ e$.

□

Una vez definida la función que elimina las observaciones deberíamos definir una equivalencia entre las configuraciones de nuestra nueva semántica y la semántica original de Sestoft que indique que ambas son esencialmente la misma, es decir, no debemos entender dicha equivalencia como una equivalencia matemática que mantendría las propiedades reflexiva, transitiva y simétrica, que la que presentaremos no las mantendrá. Uno de los problemas a abordar es que en nuestras reglas semánticas se introducen nuevos punteros y, por tanto, las expresiones que aparecen en ambas reglas contienen nuevos nombres de punteros distintos entre ambas semánticas. Este problema ya ha sido abordado en muchos trabajos [Pit05, UPG04, MFH95] y consiste en un problema de equivalencia de grafos. La solución más sencilla a este problema de nombres suele ser utilizar una α-equivalencia entre ambas configuraciones, la de Sestoft y la nuestra. En este caso, sin embargo, la solución es un tanto más compleja ya que nuestras reglas semánticas $Var@C$ y $App@$ añaden nuevos punteros al *heap*. Estos punteros apuntan a los punteros originales, pero están marcados indicando que están siendo observados, es decir, se pueden considerar como punteros "de paso" para realizar las observaciones. Al eliminar las observaciones de nuestra configuración, dichos punteros se mantendrán y se corresponderán con una especie de indirección. Por tanto, nuestra equivalencia entre configuraciones $H : e \ \equiv_{\mathsf{R}}\ H' : e'$ deberá tener en cuenta tanto estos punteros, como que los nombres de las variables libres de la expresión son distintos. Una vez definida dicha

equivalencia diremos que $H : w \equiv_{\mathsf{R}} H' : w'$ si w en H tienen el mismo valor que w' en H', es decir, si se obtiene la misma expresión siguiendo los punteros. Dicha equivalencia se puede realizar de la siguiente manera: si p es un puntero en w y $(p \mapsto e) \in H$, sustituimos todas las apariciones de p en w por e. Realizando esta operación, se obtiene una nueva expresión que puede contener punteros; en ese caso, repetimos el proceso. Análogamente, se puede realizar la misma operación sobre w'. Si ambos procesos finalizan entonces con observar si ambas expresiones tienen el mismo valor, es decir, son la misma, ya habríamos terminado. El problema aparece cuando uno de los procesos no termina. Entonces tenemos que tener en cuenta el *límite* de ambas secuencias: w y w' tienen el mismo valor si una secuencia es una subsecuencia de la otra.

Definición 6.2 Sea e una expresión y H un *heap*.

- Denotaremos por $rp\ e$ la sustitución de todos los punteros en e por el símbolo $\perp$.
- Denotaremos $H\ e$ la aplicación del *heap* H a la expresión e de Sestoft. Se define de forma recursiva, y todos los casos son triviales excepto el del puntero:

$$
\begin{array}{lcll}
H\ p & \stackrel{\text{def}}{=} & rp\ e & \text{si } (p \mapsto e) \in H \\
H\ p & \stackrel{\text{def}}{=} & \perp & \text{si } (p \mapsto e) \notin H \\
H\ \lambda\ x.e & \stackrel{\text{def}}{=} & \lambda\ x.H\ e & \\
H\ p\ q & \stackrel{\text{def}}{=} & H\ p\ H\ q & \\
H\ \textbf{letrec}\ \overline{x_i\ =\ e_i}\ \textbf{in}\ e & \stackrel{\text{def}}{=} & \textbf{letrec}\ \overline{x_i\ =\ H\ e_i}\ \textbf{in}\ H\ e & \\
H\ (C\ \overline{q_i}) & \stackrel{\text{def}}{=} & C\ \overline{H\ q_i} & \\
H\ (\textbf{case}\ p\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to e_i}) & \stackrel{\text{def}}{=} & \textbf{case}\ p\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \to H\ e_i} & \\
H\ prim & \stackrel{\text{def}}{=} & prim & \\
H\ (op\ \overline{q_i}) & \stackrel{\text{def}}{=} & op\ \overline{H\ q_i} &
\end{array}
$$

□

Definición 6.3 Sean $H : e$, $H' : e'$ dos configuraciones. Consideraremos las siguientes secuencias posiblemente infinitas

$$s = [e, H\ e, H^2\ e, H^3\ e, \ldots] \qquad \text{y} \qquad s' = [e', H'\ e', H'^2\ e', H'^3\ e', \ldots]$$

Diremos que:

- $H : e \equiv H' : e'$ si
  - $rp\ e = rp\ e'$
  - $\forall i\, \exists j \geq i,\ rp\ H^i\ e = rp\ H'^j\ e'$
  - $\forall j,\ rp\ H'^j\ e' \neq rp\ H'^{j+1}\ e' \Rightarrow \exists i \leq j,\ rp\ H^i\ e = rp\ H'^j\ e'$
- $H : e \equiv_{\mathsf{R}} H' : e'$ si $H : e \equiv \mathsf{R}\ (H' : e')$

□

De acuerdo con la definición previa, si $H : e \equiv_{\mathsf{R}} H' : e'$ sabremos que las expresiones e y e' son esencialmente las mismas, y la única diferencia que existe entre ambas son los punteros extra. Obsérvese que lo primero que se requiere es que $rp\ e = rp\ e'$: si e es una lambda expresión, una aplicación, una expresión **letrec**, un constructor o una expresión **case** entonces e' también, y viceversa; la expresión es la misma. Con esta definición de equivalencia, no se requiere que los punteros sean los mismos en ambas expresiones, pero sí se requiere que para cada puntero p de la expresión e, $(p \mapsto e_1) \in H$, haya una secuencia de punteros $[q_1 \mapsto q_2, \ldots q_n \mapsto e_n] \subseteq H'$ tal que q_1 aparezca en e' y si se aplican las substituciones correspondientes en e y e' entonces se obtiene la misma expresión. Téngase en cuenta que con esta definición no es necesario utilizar un α-renombramiento, ya que los punteros son eliminados para comprobar la equivalencia.

Una vez definida la equivalencia entre las configuraciones de ambas semánticas, debemos probar que la evaluación de una expresión marcada y su equivalente sin marcas de observación nos llevan a formas normales equivalentes. El siguiente teorema establece dicha propiedad:

Teorema 6.1 *Para toda expresión $e \in Sestoft$ y toda expresión $e^{@} \in Sestoft^{@}$ tal que $e = \mathbb{R}\ e^{@}$ entonces:*

$$\{\,\} : e^{@} \looparrowright \langle\,\rangle \ \Downarrow_{\{\,\},\{\,\}}\ K^{@} : w^{@} \looparrowright f \ \textit{sii}\ \{\,\} : e \ \Downarrow_{\{\,\},\{\,\}}\ K : w$$

$$y\ K : w \equiv_{\mathsf{R}}\ K^{@} : w^{@}$$

Para probar este teorema de forma sencilla debemos tener en cuenta algunas consideraciones. La primera, es que debemos sustituir la regla *Var* con esta otra:

$$\frac{H : e \looparrowright f \ \Downarrow_{A,C}\ K : w \looparrowright f'}{H[p \mapsto e] : p \looparrowright f \ \Downarrow_{A,C}\ K \diamond [p \mapsto w] : w \looparrowright f'} \qquad \textit{Var}'$$

tal y como se comentó en la Sección 2.1.3, a nivel semántico es lo mismo eliminar la clausura del *heap* que mantenerla en él.

La única diferencia entre ambas reglas es que la original eliminaba la clausura bajo evaluación y la nueva regla la mantiene. Durante la evaluación de la expresión e, $(p \mapsto e)$ se mantiene en el *heap*. Además, en este caso la notación $K \diamond [p \mapsto w]$ significa que actualizamos en el *heap* K la expresión correspondiente con el puntero p con la expresión w. Es muy fácil probar la equivalencia entre ambas reglas. Si en la evaluación de $H : e$ la clausura $(p \mapsto e)$ tiene que ser utilizada para llegar a reducirla entonces se entra en lo que se llama un "agujero negro", porque se necesita p para evaluar p. En este caso, con la nueva regla *Var'* la evaluación no finalizará, mientras que con la regla *Var* la evaluación hubiera parado sin llegar a una forma normal, es decir, en ningún caso tiene semántica.

Un detalle interesante que se desprende de la modificación de dicha regla, es que el conjunto A deja de ser imprescindible para mantener la frescura comprobable de forma local. Esto se debe a que a partir de este momento las clausuras bajo evaluación se mantienen en el *heap*. Gracias a esto la demostración de equivalencia entre ambas semánticas se realizará de forma más sencilla, ya que no tendremos que tener en cuenta el conjunto A. Recordemos que el conjunto C sólo es necesario para la definición de frescura local en cada regla.

Proposición 6.2 $H : e \looparrowright f \ \Downarrow_{A,C}\ K : w \looparrowright f'$ *sii* $H : e \looparrowright f \ \Downarrow_{A,C}\ K : w \looparrowright f'$ *con la regla* $\textit{Var}'$.

Demostración 2 La prueba se realiza por inducción. Todas las reglas son triviales excepto la prueba de la regla *Var*. La implicación $\Rightarrow$ es inmediata ya que con la regla Var' no se eliminan las clausuras bajo evaluación y si eliminándolas alcanzamos una forma normal, sin eliminarlas también. Nos centraremos en la implicación $\Leftarrow$.

Sabemos que $H : e \twoheadrightarrow f \ \Downarrow_{A,C} \ K : w \twoheadrightarrow f'$, por tanto dos alternativas pueden haber sucedido. Consideraremos como primer caso aquél en el que la clausura $p \mapsto e$ no ha sido utilizada en la derivación, por tanto podemos aplicar hipótesis de inducción sin problema y la demostración finaliza. El segundo caso se corresponde al caso donde la clausura ha sido utilizada, pero en este caso llegaremos a un absurdo. Para reducir p ha sido necesario reducir p, esto nos lleva a que se ha producido un ciclo, por tanto es imposible $H : e \twoheadrightarrow f \ \Downarrow_{A,C} \ K : w \twoheadrightarrow f'$; ya que en dicho punto del ciclo hemos vuelto a evaluar $H[p \mapsto e] : p$. En este punto, podemos afirmar que acabamos de entrar en un bucle sin fin y por tanto la evaluación de p no puede alcanzar una forma normal.

En lo que resta de esta sección, consideraremos que hemos utilizado la regla Var' en vez de la regla *Var*. Antes de probar el resultado principal, necesitamos ciertas propiedades que se mantienen invariables a lo largo de la evaluación.

Definición 6.4 $H : e$ es una configuración *buena* si todos los punteros alcanzables se encuentran ligados en el *heap*.

□

Proposición 6.3 *Sea $H : e$ una configuración buena. Si $H : e \twoheadrightarrow f \ \Downarrow_{A,C} \ K : w \twoheadrightarrow f'$ entonces $K : w$ y $K : e$ son configuraciones buenas.*

Demostración 3 La demostración se realiza por inducción sobre las reglas. Dicha prueba es muy simple, ya que sólo es necesario tener en cuenta las reglas que eliminan o crean nuevas clausuras. En este caso, debido a la modificación de la regla, Var' ya no elimina las clausuras del *heap*. Además, las reglas *Letrec* y $App@$ son las únicas que crean nuevas clausuras en el *heap* y mantienen la propiedad debido a la definición de frescura.

Para demostrar el teorema principal es necesario previamente tener en consideración algunas de las propiedades que las relaciones $\equiv$ y $\equiv_{\mathsf{R}}$ satisfacen. Su demostración es muy sencilla, ya que sólo consiste en considerar la definición y en ciertos casos aplicar inducción sobre las reglas.

Propiedad 6.1 *Sean $H : e$, $H' : e'$ y $H'' : e''$ tres configuraciones semánticas, entonces:*

1. *Si $H : e \equiv H' : e'$ y $H' : e' \equiv H'' : e''$ entonces $H : e \equiv H'' : e''$*
2. *Si $H : e \equiv H' : e'$ entonces $\mathsf{R}\ (H : e) \equiv_{\mathsf{R}} \ H' : e'$*
3. *Si $H : e \equiv H' : e'$ y $H' : e' \equiv_{\mathsf{R}} \ H'' : e''$ entonces $H : e \equiv_{\mathsf{R}} \ H'' : e''$*
4. *Si $H : e \equiv_{\mathsf{R}} \ H' : e'$ entonces si q' es fresca $\forall (q \mapsto e') \in H'$, $H : e \equiv_{\mathsf{R}} \ H' \cup [q' \mapsto q^{@}] : \ e'\ [q'/q]$*
5. *Si $H : e \equiv_{\mathsf{R}} \ H' : e'$ entonces si $\overline{q_i}$ son frescas $H : e \equiv_{\mathsf{R}} \ H' \cup [\overline{q_i \mapsto q_{i-1}^{@}}^{n}] : \ q_n^{@}$, $n \geq 0$ y $q_0 = e'$*
6. *Si $H : \lambda x.e \equiv_{\mathsf{R}} \ H' : \lambda^{@} x.e^{@}$ y $H : p \equiv_{\mathsf{R}} \ H' : p'$ entonces $H : e[p/x] \equiv_{\mathsf{R}} \ H' : e^{@}[p'/x]$*

7. *Sea* $H[\overline{q_i \mapsto q_{i-1}^{@}}^{n}] : \ q_n,\ n \geq 0\ y\ q_0 = e$
   Si $H : e \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f'$ *entonces* $H : q_n \rightarrowtail f \ \Downarrow_{A,C} \ K \cup [\overline{q_i \mapsto w^i}^{n}, \overline{q_i' \mapsto q''^{@}_i}] : \ w^0 \rightarrowtail f'$,
   $\forall i \ \exists j, \ w = (\mathsf{R}\ K)^j w^i$ *y* $\overline{q_i'}$ *son frescas.*

Usando estas propiedades, probaremos una proposición que es más general que el Teorema 6.1 original. Además de la equivalencia entre ambas semánticas, deberemos probar la siguiente propiedad: $K : e' \equiv_{\mathsf{R}} \ K^{@} : e'^{@}$ para cualquier expresión e'. Este resultado auxiliar es necesario para la prueba del teorema, ya que en la demostración de las reglas que poseen dos discriminantes (ej. *App*) en un momento de la evaluación nos olvidamos de parte de la expresión original (ej. en *App* nos olvidamos del argumento) y posteriormente lo utilizamos para terminar la reducción. Sobre los punteros que aparecen en dicha parte de la expresión original debemos mantener la equivalencia, para que cuando los utilicemos en la segunda parte de la regla se mantenga la equivalencia general.

Proposición 6.4 *Sean* $H : e$ *y* $H^{@} : e^{@}$ *dos configuraciones buenas de Sestoft y Sestoft*$^{@}$ *respectivamente, tales que* $H : e \equiv_{\mathsf{R}} \ H^{@} : e^{@}$, *sea* f *un fichero; para toda* e' *y* $e'^{@}$ *expresiones de Sestoft y Sestoft*$^{@}$ *respectivamente, tales que* $H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@}$ *entonces:*

$$H^{@} : e^{@} \rightarrowtail f \ \Downarrow_{A,C} \ K^{@} : w^{@} \rightarrowtail f' \text{ sii } H : e \ \Downarrow_{A,C} \ K : w$$
$$K : w \equiv_{\mathsf{R}} \ K^{@} : w^{@} \ y \ K : e' \equiv_{\mathsf{R}} \ K^{@} : e'^{@}$$

Demostración 4 Aquí solo presentaremos un esquema de la prueba. El lector interesado encontrará la prueba completa en el apéndice de esta tesis.

Para no complicar innecesariamente la demostración, eliminaremos el fichero de observaciones de las reglas, ya que dicho fichero no participa en la evaluación de la expresión. Este fichero se puede considerar únicamente como un efecto lateral. En las reglas que usan la longitud de dicho fichero utilizaremos cualquier entero. Nótese que esto sólo modificará las observaciones que se producen sobre el fichero, sin modificar en ningún caso la evaluación.

La prueba se realiza por inducción sobre las reglas. Nótese que si las configuraciones son *buenas*, el segundo caso considerado en la Definición 6.2 ($(p \mapsto e) \notin H$) no puede suceder. Sin embargo, la definición debe tenerlo en cuenta para mantener la completitud.

Finalmente, como corolario inmediato tenemos que el teorema 6.1 se cumple.

La última cuestión importante a demostrar es que de alguna manera la traza que observamos se corresponde con lo que hemos anotado como observable y se ha evaluado, es decir, con lo que intuitivamente queremos observar. El siguiente teorema establece dicha propiedad:

Teorema 6.2

1. *Si para la evaluación de una expresión es necesario evaluar una variable observada entonces se produce en el fichero la observación del resultado de dicha variable.*

2. *Si en el fichero hay una marca de observación, ésta se ha producido porque se ha demandado la evaluación de una variable observada.*

Demostración 5

1. La demostración de esta propiedad es algo tan sencillo como que las únicas reglas que evalúan dichas expresiones son las que aparecen en la Figura 6.3 y que dichas reglas generan en el fichero una marca de observación.

2. La demostración, al igual que en el caso anterior, se realiza de forma trivial indicando que las únicas reglas que añaden observaciones en el fichero son las relativas a la reducción de la marcas observadas.

A partir de este momento, en que ya sabemos que la semántica es correcta y que las observaciones se realizan de manera adecuada, vamos a construir partiendo de dicha semántica una máquina abstracta equivalente.

6.3. Máquina abstracta

En esta sección crearemos una máquina abstracta equivalente a la semántica mostrada anteriormente. Para ello, extenderemos una de las máquina abstractas creada por Sestoft en [Ses97]. En ese trabajo, él introdujo varias máquinas abstractas en secuencia, llamadas *Mark-1*, *Mark-2* y *Mark-3*. Utilizaremos la máquina *Mark-2* para derivar nuestra máquina porque es suficientemente cercana a la realidad y los detalles de bajo nivel son escasos.

En la Figura 2.5 se muestran las reglas operacionales de la máquina original *Mark-2*. Como se comentó en el Capítulo 2.1.3, Sestoft demostró que las derivaciones exitosas de la máquina son exactamente las mismas que las de la semántica. La razón por la que son necesarios los entornos es debida a que las expresiones de control, las λ-abstracciones y las alternativas mantienen sus variables originales y, por tanto, en tiempo de ejecución necesitamos conocer los punteros asociados con dichas variables. Básicamente, la máquina consiste en un aplanamiento del árbol semántico.

A la hora de incluir observaciones en nuestra máquina, es necesario realizar algunas modificaciones a la máquina original. La primera de ellas consiste en añadir una nueva columna en la que se observarán los efectos laterales que provocan las observaciones, es decir, las modificaciones del fichero de observaciones. En la Figura 6.4 se presentan las reglas de evaluación de una expresión sin observaciones. Se ha añadido el fichero de observaciones, pero dichas reglas no modifican este fichero, ya que dichas reglas no trabajan con las observaciones y, por tanto, no generan ninguna marca de observación.

Definición 6.5 Una *configuración* de la máquina abstracta *Mark-2*$^{@}$ es una quíntupla de la forma (H, e, E, S, f) donde H representa el *heap*, e se corresponde con la expresión de control, E es el entorno, S es la pila y f el fichero:

- El entorno E contiene las ligaduras de las variables libres de la expresión de control e con los punteros correspondientes o valores primitivos. Este entorno puede verse como una sustitución retrasada, pendiente de ser realizada.

- El *heap* H liga los punteros con las clausuras. Las clausuras son un par (e, E) donde e se corresponde con una expresión y E representa el entorno de dicha expresión, que liga las variables libres de la expresión e con sus correspondientes punteros.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	H	$x\ atom$	E	S	f	*app1*([1])
$\Longrightarrow$	H	x	E	$atom_1 : S$	f	
	H	$\lambda y.e$	E	$atom : S$	f	*app2*
$\Longrightarrow$	H	e	$E \cup \{y \mapsto atom\}$	S	f	
	$H \cup [p \mapsto (e, E_1)]$	x	$E\{x \mapsto p\}$	S	f	*var1*
$\Longrightarrow$	H	e	E_1	$\#p : S$	f	
	H	$\lambda y.e$	E	$\#p : S$	f	*var2*
$\Longrightarrow$	$H \cup [p \mapsto (\lambda y.e, E\|)]$	$\lambda y.e$	E	S	f	
	H	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	E	S	f	*letrec*([2])
$\Longrightarrow$	$H \cup \overline{[p_i \mapsto (e_i, E_1\|)]}$	e	E_1	S	f	
	H	**case** e **of** $alts$	E	S	f	*case1*
$\Longrightarrow$	H	e	E	$(alts, E) : S$	f	
	H	$C_k\ \overline{atom_i}$	E	$(alts, E_1) : S$	f	*case2*([3])
$\Longrightarrow$	H	e_k	$E_1 \cup \overline{\{y_{ki} \mapsto atom'_i\}}$	S	f	
	H	$C\ \overline{atom_i}$	E	$\#p : S$	f	*var3*
$\Longrightarrow$	$H \cup [p \mapsto (C\ \overline{atom_i}, E\|)]$	$C\ \overline{x_i}$	E	S	f	
	H	$op\ x_1\ x_2$	$E\{\overline{x_i \mapsto m_i}^2\}$	S	f	*op#*
$\Longrightarrow$	H	$m_1\ op\ m_2$	$\{\,\}$	S	f	

([1]) $atom_1 =$ si $isPrim(atom)$ entonces $atom$ sino $(E\ atom)$
([2]) $\overline{p_i}$ son distintas y frescas con respecto a H, **letrec** $\{\overline{x_i = e_i}\}$ **in** e y S y $E_1 = E \cup \{\overline{x_i \mapsto p_i}\}$.
([3]) e_k se corresponde con la alternativa $C_k\ \overline{y_{ki}} \to e_k$ en $alts$,
$atom'_i =$ si $isPrim(atom_i)$ entonces $atom_i$ sino $(E\ atom_i)$

Figura 6.4: Máquina abstracta *Mark-2* adaptada

- La pila S almacena tres tipos de objetos y su sintaxis es la siguiente:

$$\begin{array}{rcl} S & \to & \overline{p_i} : S \\ & | & (\overline{alt_i}, E) : S \\ & | & \#p : S \\ & | & [\,] \end{array}$$

  - $\overline{p_i}$ se corresponde con un conjunto de argumentos pendientes de aplicación,
  - $(\overline{alt_i}, E)$ se corresponde con las alternativas de las expresiones **case** pendientes de realizar el encaje de patrones cuando la expresión bajo evaluación alcance la forma normal débil de cabeza, junto con su entorno E,
  - $\#p$ se corresponde con las marcas de actualización de las ligaduras $[p \mapsto (e, E)]$ que actualmente se encuentran bajo evaluación.

- El fichero f almacena las observaciones que se producen a lo largo del cómputo.

□

Siguiendo las mismas ideas que Sestoft, nosotros hemos derivado nuevas reglas en la máquina que contemplan las nuevas reglas semánticas. Es en esta derivación donde aparece la segunda modificación, que consiste en que, para manejar las observaciones de forma adecuada, tenemos

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	H	$x^{@str}$	E	S	f	$var@S$
$\Longrightarrow$	H	$x^{@(length\ f,0)}$	E	S	$f \circ \langle 0\,0\ Observe\ str \rangle$	
	H	$x^{@(r,s)}$	E	S	f	$var1@$
$\Longrightarrow$	H	x	E	$@(r,s) : S$	$f \circ \langle r\ s\ Enter \rangle$	
	H	$\lambda y.e$	E	$@(r,s) : S$	f	$var2@$
$\Longrightarrow$	H	$\lambda^{@[(r,s)]} y.e$	E	S	f	
	H	$\lambda^{obs} y.e$	E	$@(r,s) : S$	f	$var2@@$
$\Longrightarrow$	H	$\lambda^{@(r,s):obs} y.e$	E	S	f	
	H	$C\ \overline{atom_i}$	E	$@(r,s) : S$	f	$var3@(^1)$
$\Longrightarrow$	$H \cup [\overline{q_i \mapsto (atom_i^{@(length\ f,i)}, E)}]$	$C\ \overline{x_i}$	$\{\overline{x_i \mapsto q_i}\}$	S	$f \circ \langle r\ s\ Cons\ k\ C \rangle$	
	H	$\lambda^{@[\overline{(r_i,s_i)}]} y.e$	E	$p : S$	f	$app2@(^2)$
$\Longrightarrow$	$H \cup \begin{bmatrix} q \mapsto (e, E \cup \{y \mapsto q_1\}\vert), \\ q_1 \mapsto (arg^{@(length\ f,0)}, \{arg \mapsto p\}) \end{bmatrix}$	$ap^{@(length\ f,1)}$	$\{ap \mapsto q\}$	S	$f \circ \langle\, \overline{[(r_i,s_i)]}\ Fun \rangle$	

(^{1}) $\overline{q_i}$ son distintas y frescas con respecto a H, $C\ \overline{x_i}$ y S.

(^{2}) q, q_1 son distintas y frescas con respecto a H, $\lambda^{@[\overline{(r_i,s_i)}]} y.e$ y S.

Figura 6.5: Máquina abstracta *Mark-2*$^{@}$, reglas para Hood

que añadir un nuevo tipo de objeto en la pila. Este nuevo tipo de objeto, $@(r,s)$, se corresponde con las marcas pendientes de observación. Estas nuevas reglas se presentan en la Figura 6.5.

Recalcaremos los siguientes hechos:

- Los efectos laterales se producen a la vez que se realiza la evaluación del programa.
- Las observaciones se pueden obtener incluso aunque el programa no finalice su cómputo, ya que éstas se escriben en un fichero independiente.

Empezaremos a describir en detalle las nuevas reglas. La regla $var@S$ se corresponde con la regla semántica $Var@S$; la regla *var1*@ se corresponde con la primera parte de la evaluación de las reglas semánticas $Var@F$, $Var@FO$ y $Var@C$; la regla *var2*@ se corresponde con la segunda parte de la evaluación de la regla semántica $Var@F$; la regla *var2@@* se corresponde con la segunda parte de la evaluación de la regla semántica $Var@FO$; la regla *var3*@ se corresponde con la segunda parte de la evaluación de la regla semántica $Var@C$; y la regla *app2*@ se corresponde con la regla semántica $App@$. Cuando alcanzamos un constructor y en la cima de la pila se encuentra una marca de observación, regla *var3*@, realizamos la observación y continuamos con la evaluación considerando que ahora estamos observando los argumentos del constructor. La regla *var2*@ sólo se encarga de anotar la lambda con su marca de observación correspondiente. Cuando la evaluación de una expresión se corresponde con la aplicación de una λ-abstracción que se encuentra observada sobre un argumento, regla *app2*@, se realiza la observación indicando que dicha clausura evaluó a una función y se evalúa el cuerpo de la función. Hay que tener en cuenta que no sólo estamos observando el argumento de la función, sino que también observamos el resultado de dicha aplicación.

Como se puede comprobar, la generación de una máquina abstracta a partir de la semántica no parece complicada. Básicamente consiste en almacenar en una pila las continuaciones para que cuando terminemos un cómputo sepamos por dónde seguir con la evaluación. No obstante, la máquina que se genera no es eficiente. A este nivel se pueden ir añadiendo distintas optimizaciones

a dicha máquina, tales como las expresiones **otherwise** sobre los **case**, el encapsulamiento y desencapsulamiento de los primitivos, la poda de los entornos, etc. para conseguir una máquina más eficiente. De esta forma iríamos acercándonos a la eficiencia de la máquina *STG*.

6.3.1. Corrección y equivalencia

Ahora tenemos que demostrar la corrección de la nueva máquina, que llamaremos *Mark-2*$^{@}$, con respecto de la nueva semántica (Figura 6.3). Como previamente hemos demostrado que nuestra semántica y la semántica original son equivalentes, obtendremos como corolario que las máquinas *Mark-2*$^{@}$ y *Mark-2* son equivalentes.

Para demostrar la corrección debemos tener en cuenta que el *heap* semántico y el *heap* de la máquina *Mark-2*$^{@}$ poseen varias diferencias estructurales. El *heap* de la máquina *Mark-2*$^{@}$ posee entornos que vinculan las variables libres de las expresiones con sus punteros correspondientes; mientras que las expresiones del *heap* de la semántica han sustituido sus variables libres por los punteros correspondientes. Por tanto, debemos definir una equivalencia entre ellos.

Definición 6.6 Sea H un *heap* de la máquina *Mark-2*$^{@}$, H_{env} denota el siguiente *heap*: $[p \mapsto E\ e \mid (p \mapsto (e, E)) \in H]$

□

Teorema 6.3 *Para toda* $e \in$ *Sestoft*$^{@}$ *entonces:*

$$\{\,\} : e \hookrightarrow \langle\,\rangle \ \Downarrow_{\{\,\},\{\,\}}\ K : w \hookrightarrow f \ \textit{sii}\ (\{\,\}, e, \{\,\}, [\,], \langle\,\rangle) \Longrightarrow^{*} (K', w', E', [\,], f'),$$
$$K = K'_{env},\ w = E'\ w'\ y\ f = f'$$

Para demostrar este teorema y para separar las ideas concernientes a los entornos de la demostración principal, definiremos una máquina auxiliar, la máquina *Mark-1*$^{@}$, que es similar a la máquina *Mark-2*$^{@}$. La principal diferencia técnica es que no posee entornos (véase la Figura 6.6). Ahora demostraremos que la máquina *Mark-1*$^{@}$ es equivalente a la semántica natural con observaciones (Figura 6.3) y a la máquina *Mark-2*$^{@}$. Como corolario se obtiene la prueba del Teorema 6.3.

Previamente a entrar en los detalles de la demostración definiremos las configuraciones de la máquina *Mark-1*$^{@}$:

Definición 6.7 Una *configuración* de la máquina abstracta *Mark-1*$^{@}$ es una cuádrupla de la forma (H, e, S, f) donde H representa el *heap*, e se corresponde con la expresión de control, S es la pila y f el fichero:

- El *heap* H liga los punteros con expresiones.
- La pila S almacena tres tipos de objetos y su sintaxis es la siguiente:

$$\begin{array}{rcl} S & \rightarrow & \overline{p_i} : S \\ & \mid & \overline{alt_i} : S \\ & \mid & \#p : S \\ & \mid & @(r, s) : S \\ & \mid & [\,] \end{array}$$

	Heap	Control	Pila	Efecto lateral	regla	
	H	$(p\ q)$	S	f	*app1*	
$\Longrightarrow$	H	p	$q : S$	f		
	H	$\lambda y.e$	$p : S$	f	*app2*	
$\Longrightarrow$	H	$e[p/y]$	S	f		
	$H \cup [p \mapsto e]$	p	S	f	*var1*	
$\Longrightarrow$	H	e	$\#p : S$	f		
	H	$\lambda y.e$	$\#p : S$	f	*var2*	
$\Longrightarrow$	$H \cup [p \mapsto \lambda y.e]$	$\lambda y.e$	S	f		
	H	**letrec** $\{\overline{x_i = e_i}\}$ **in** e	S	f	*letrec*	(^{1})
$\Longrightarrow$	$H \cup [\overline{p_i \mapsto \hat{e}_i}]$	$\hat{e}$	S	f		
	H	**case** p **of** *alts*	S	f	*case1*	
$\Longrightarrow$	H	p	$alts : S$	f		
	H	$C_k\ \overline{p_i}$	$alts : S$	f	*case2*	(^{2})
$\Longrightarrow$	H	$e_k[\overline{p_i/y_{ki}}]$	S	f		
	H	$C_k\ \overline{p_i}$	$\#p : S$	f	*var3*	
$\Longrightarrow$	$H \cup [p \mapsto C_k\ \overline{p_i}]$	$C_k\ \overline{p_i}$	S	f		
	H	$op\ m_1\ m_2$	S	f	*op#*	
$\Longrightarrow$	H	$m_1\ op\ m_2$	S	f		
	H	$p^{@str}$	S	f	*var@S*	
$\Longrightarrow$	H	$p^{@(n,0)}$	S	$f \circ \langle 0\,0\ Observe\ str \rangle$		
	H	$p^{@(r,s)}$	S	f	*var1@*	
$\Longrightarrow$	H	p	$@(r,s) : S$	$f \circ \langle r\ s\ Enter \rangle$		
	H	$\lambda y.e$	$@(r,s) : S$	f	*var2@*	
$\Longrightarrow$	H	$\lambda^{@(r,s)} y.e$	S	f		
	H	$\lambda^{obs} y.e$	$@(r,s) : S$	f	*var2@@*	
$\Longrightarrow$	H	$\lambda^{@(r,s):obs} y.e$	S	f		
	H	$C\ \overline{p_i}^k$	$@(r,s) : S$	f	*var3@*	(^{3})
$\Longrightarrow$	$H \cup [\overline{q_i \mapsto p_i^{@(n',i)}}]$	$C\ \overline{q_i}$	S	$f \circ \langle r\ s\ Cons\ k\ C \rangle$		
	H	$\lambda^{@[\overline{(r_i,s_i)}]} y.e$	$p : S$	f	*app2@*	(^{4})
$\Longrightarrow$	$H \cup \left[\begin{array}{l} q \mapsto e[q_1/y], \\ q_1 \mapsto p^{@(n',0)} \end{array}\right]$	$q^{@(n',1)}$	S	$f \circ \langle\, [\overline{(r_i,s_i)}]\ Fun \rangle$		

(^{1}) $\overline{p_i}$ son distintas y frescas respecto a H, **letrec** $\{\overline{x_i = e_i}\}$ **in** e y S. $\hat{e} = e[\overline{p_i/x_i}]$
(^{2}) e_k se corresponde con la alternativa $C_k\ \overline{y_{ki}} \to e_k$ de *alts*
(^{3}) $\overline{q_i}$ son distintas y frescas respecto a H, $C\ \overline{p_i}$ y S.
(^{4}) q, q_1 son distintas y frescas respecto a H, $\lambda^{@[\overline{(r_i,s_i)}]} y.e$ y S.

Figura 6.6: Máquina abstracta *Mark-1*$^{@}$

- $\overline{p_i}$ se corresponde con un conjunto de argumentos pendientes de aplicación,
- $\overline{alt_i}$ se corresponde con las alternativas de las expresiones **case** pendientes de realizar el encaje de patrones cuando la expresión bajo evaluación alcance la forma normal débil de cabeza,
- $\#p$ se corresponde con las marcas de actualización de las ligaduras $[p \mapsto e]$ que actualmente se encuentran bajo evaluación.

□

Ahora, demostraremos la equivalencia entre ambas máquinas. Para demostrar dicha equivalencia definiremos una equivalencia entre las configuraciones de la máquina *Mark-1*$^{@}$ y la máquina *Mark-2*$^{@}$.

Definición 6.8

1. Sea S una pila de la máquina *Mark-2*$^{@}$ entonces S_{env} denota la pila siguiente:

$$\begin{cases} p : S'_{env} & \text{si } S = p : S'; \\ \#p : S'_{env} & \text{si } S = \#p : S'; \\ @(r,s) : S'_{env} & \text{si } S = @(r,s) : S'; \\ E\ alts : S'_{env} & \text{si } S = (alts, E) : S'; \\ [\,] & \text{si } S = [\,]; \end{cases}$$

2. Sea (H, e, E, S, f) una configuración de la máquina *Mark-2*$^{@}$ entonces:

   $(H, e, E, S, f)_{env}$ denota la configuración de la máquina *Mark-1*$^{@}$ $(H_{env}, E\ e, S_{env}, f)$

3. Sea (H, e, S, f) una configuración de la máquina *Mark-1*$^{@}$ y (H', e', E', S', f') una configuración de la máquina *Mark-2*$^{@}$ entonces $(H, e, S, f) \equiv_{env} (H', e', E', S', f')$ si:

$$(H, e, S, f) = (H', e', E', S', f')_{env}$$

□

La propiedad siguiente establece la equivalencia entre ambas máquinas:

Proposición 6.5 *Dada una configuración (H, e, S, f) de la máquina Mark-1*$^{@}$ *y una configuración (H', e', E', S', f') de la máquina Mark-2*$^{@}$ *tal que $(H, e, S, f) \equiv_{env} (H', e', E', S', f')$ entonces:*
$(H, e, S, f) \Longrightarrow^{*} (K, w, S, f)$ sii $(H', e', E', S', f') \Longrightarrow^{*} (K', w', E', S', f'')$ y $(K, w, S, f) \equiv_{env} (K', w', E', S', f'')$.

Demostración 6 La demostración se realiza pon inducción sobre las reglas. Es trivial, ya que todas las reglas son equivalentes via $\equiv_{env}$.

Ahora, para demostrar la equivalencia entre la nueva máquina y las reglas semánticas, seguiremos las mismas ideas presentadas por Sestoft. Para ello será necesario introducir el concepto de *trazas equilibradas*. Primero nos concentraremos en la máquina abstracta sin observaciones. Intuitivamente, una traza balanceada se corresponde con la evaluación de una expresión completa, pero comenzando con una pila posiblemente no vacía. En este contexto la *traza* es una secuencia de reglas de la máquina abstracta. Las características principales de una traza balanceada son:

- La pila original permanece sin ser modificada. Ninguna de las reglas aplicadas en una traza balanceada cambia ni incluso consulta un valor en dicha parte de la pila. Las reglas pueden apilar y desapilar nuevos valores en la cima de la pila original.
- Al final de la ejecución de la traza la pila resultante es la original. Ningún valor extra puede encontrarse en la cima de la pila cuando la traza finaliza, la pila final es la misma que la inicial.

Como una traza balanceada se corresponde con la evaluación completa de una expresión, cualquier traza balanceada no vacía debe comenzar con la regla *app1*, *var1* o *letrec*. Una traza balanceada no puede comenzar con las reglas *app2* o *var2* ya que en ambos casos en la cima de la pila se encuentra un argumento pendiente que indica que la evaluación de la expresión había comenzado previamente.

Si la traza comienza con *app1*, produce una pila intermedia de la forma $p : S$. La única regla que elimina dicho puntero de la pila y reestablece la pila es la regla *app2*, por tanto debe aparecer en la traza. A partir de ese momento en el control tenemos una expresión e, por tanto la subtraza entre *app1* y su correspondiente *app2* es una traza balanceada. Análogamente, si la traza comienza con la regla *var1* debe terminar con la regla *var2* o la regla *var3*; si comienza con la regla *case1*, *case2* aparece eventualmente (en este caso la evaluación debe continuar con la expresión correspondiente a la alternativa).

Por tanto, en la máquina abstracta *Mark-1*, las trazas balanceadas se corresponden con las trazas que se derivan de la siguiente gramática:

$$\begin{aligned} bal ::= \; & \epsilon \mid app1 \;\; bal \;\; app2 \;\; bal \mid var1 \;\; bal \;\; var2 \mid letrec \;\; bal \mid \\ & var1 \;\; bal \;\; var3 \mid case1 \;\; bal \;\; case2 \;\; bal \mid op\# \end{aligned}$$

Ahora volvamos a nuestra máquina *Mark-1*$^@$. La nueva regla $var@S$ no modifica la pila, por tanto, $var@S$ *bal* es una traza balanceada que se corresponde con la regla semántica $Var@S$. Nuestra regla *var1*@ sigue el mismo razonamiento que la regla *var1*, entonces *var1*@ *bal var2*@, *var1*@ *bal var2*@@ y *var1*@ *bal var3*@ son también trazas balanceadas que se corresponden con las reglas semánticas $Var@F$, $Var@FO$ y $Var@C$ respectivamente. La traza vacía, ϵ, se corresponde con las reglas *Cons*, *Lam* o *Lam*@.

Definición 6.9

- Una *traza balanceada* es una secuencia de reglas que puede ser derivada de la siguiente gramática:

$$\begin{aligned} bal ::= \; & \epsilon \mid app1 \;\; bal \;\; app2 \;\; bal \mid var1 \;\; bal \;\; var2 \mid var1 \;\; bal \;\; var3 \mid letrec \;\; bal \mid \\ & case1 \;\; bal \;\; case2 \;\; bal \mid op\# \mid app1 \;\; bal \;\; app2@ \;\; bal \mid var@S \;\; bal \mid \\ & var1@ \;\; bal \;\; var2@ \mid var1@ \;\; bal \;\; var2@@ \mid var1@ \;\; bal \;\; var3@ \end{aligned}$$

- Sean (H_1, e_1, S_1, f_1) y (H_2, e_2, S_2, f_2) dos configuraciones de la máquina *Mark-1*$^@$; diremos que el cómputo
$$(H_1, e_1, S_1, f_1) \Longrightarrow^* (H_2, e_2, S_2, f_2)$$
es *balanceado* si $S_1 = S_2$ y ninguna de las reglas involucradas en el cómputo consulta ningún valor en la pila S_1.

□

Entonces, es fácil demostrar que un cómputo balanceado de la máquina *Mark-1*$^@$ se corresponde con una traza balanceada.

Proposición 6.6 *Sean* (H_1, e_1, S_1, f_1) *y* (H_2, e_2, S_2, f_2) *dos configuraciones de la máquina Mark-1*$^@$*. El cómputo*
$$(H_1, e_1, S_1, f_1) \Longrightarrow^* (H_2, e_2, S_2, f_2)$$
es balanceado sii la secuencia de las reglas aplicadas es balanceada.

Demostración 7 Veremos la demostración de ambas implicaciones por separado, ya que son bastante diferentes.

$\Longrightarrow$ La discusión previa muestra que cualquier traza balanceada debe comenzar con una de las reglas siguientes:

app1. El cómputo balanceado tendrá la forma: *app1 bal app2 bal* o *app1 bal app2@ bal*.

letrec. El cómputo balanceado tendrá la forma: *letrec bal*.

case1. El cómputo balanceado será: *case1 bal case2 bal*.

op#. El cómputo balanceado será: *op#*.

var@S. El cómputo balanceado será: *var@S bal*.

var1 **o** *var1@*. En este caso, ponemos un elemento en la cima de la pila; dicho elemento debe ser eliminado para conseguir que el cómputo sea balanceado. Las reglas que eliminan dicho elemento se corresponden con las siguientes: *var2*, *var3*, *var2@*, *var2@@* o *var3@*. El cómputo entre ambas reglas debe corresponder con un cómputo balanceado y, por tanto con una traza balanceada. Este cómputo puede continuar con otro cómputo balanceado. Pero como el control tras la regla *var2* y *var2@* se corresponde con una λ-abstracción, y tras la regla *var3* y *var3@* con un constructor, sólo una regla que se encargue de leer la cima de la pila puede ser aplicada. Si ése fuera el caso, el cómputo no sería balanceado. Por tanto, la traza balanceada debe finalizar con la regla *var2*, *var3*, *var2@*, *var2@@* o *var3@*.

$\Longleftarrow$ La prueba se realiza por inducción estructural en la forma de la traza balanceada. El caso base de la inducción se corresponde con la traza ϵ cuya demostración es trivial, ya que no existe ningún cómputo. Todos los casos inductivos son similares, por tanto pondremos de ejemplo el caso de la traza *app1 bal app2@ bal*.

En este caso tenemos que $e_1 = e'\ x$, por tanto;
$$(H_1, e\ x, S_1, f_1) \Longrightarrow (H', e', p : S, f')$$

que cumple los requerimientos de la proposición. Entonces, por inducción estructural tenemos un cómputo que cumple la tesis de esta proposición, es decir, tenemos que

$$(H', e', p : S, f') \Longrightarrow^* (H'', e'', p : S, f'')$$

Entonces ejecutamos la regla *app2@*:

$$(H'', e'', p : S, f'') \Longrightarrow (H''', e''', S, f''')$$

que cumple la proposición. Finalmente, por inducción estructural, tenemos un cómputo que cumple la tesis de la proposición:

$$(H''', e''', S, f''') \Longrightarrow^* (H_2, w_2, S_2, f_2)$$

La proposición siguiente demuestra la equivalencia entre la semántica y la máquina *Mark-1*$^{@}$.

Proposición 6.7 *Sea H un heap, $e \in$ Sestoft$^{@}$, S una pila, f un fichero entonces:*
$H : e \looparrowright f \;\Downarrow\; K : w \looparrowright f'$ *sii* $(H, e, S, f) \Longrightarrow^* (K, w, S, f')$ *es balanceada.*

Demostración 8 La demostración completa se puede encontrar en el apéndice de esta tesis y consiste en una extensión de la prueba que aparece en [Ses97], con la única diferencia que ahora tenemos que tener en cuenta las nuevas reglas que trabajan con las observaciones y el fichero en el que se almacenan dichas observaciones.

Como corolario de la Proposición 6.7 y la Proposición 6.5 se obtiene el siguiente Teorema 6.3. Además, como corolario de los Teoremas 6.1 y 6.3 y los teoremas de equivalencia de Sestoft (véase [Ses97]) obtenemos la equivalencia entre la máquina original *Mark-2* y nuestra máquina con observaciones *Mark-2*$^{@}$.

Corolario 6.1 *Para toda $e \in$ Sestoft, $e^{@} \in$ Sestoft$^{@}$ tal que $e =$ R $e^{@}$ entonces:*

$$(\{\,\}, e, \{\,\}, [\,]) \Longrightarrow^* (K, w, E, [\,]) \text{ es una derivación de la máquina Mark-2}$$
$$\text{sii}$$
$$(\{\,\}, e^{@}, \{\,\}, [\,], \langle\,\rangle) \Longrightarrow^* (K^{@}, w^{@}, E^{@}, [\,], f) \text{ es una derivación de la máquina Mark-2}^{@}$$
$$\text{y } K_{env} : E\ w \equiv_{\mathsf{R}}\ K^{@}_{env} : E^{@}\ w^{@}$$

6.3.2. Ejemplo de evaluación de una expresión en la máquina abstracta

Ya se ha visto que la máquina *Mark-2*$^{@}$ y la semántica son equivalentes, por tanto los ejemplos vistos en la Sección 6.2.1 evalúan en nuestra máquina obteniendo los mismos resultados que en la semántica. Por tanto, consideramos que no parece muy interesante desarrollar varios ejemplos sobre la máquina abstracta. Así pues, desarrollaremos únicamente un ejemplo completo para que se vean los cómputos de paso corto que realiza la máquina.

Veremos la reducción en la máquina del primero de los ejemplos que vimos en la semántica. En la semántica se observaron todas las reducciones de dicho ejemplo, ya que era un ejemplo sencillo, por tanto, ahora en la máquina haremos lo mismo.

Ejemplo 6.6 El ejemplo, partía de la siguiente expresión inicial de Haskell:

```
observe "obs2" (observe "obs1" (10::Int))::Int
```

que en nuestro lenguaje tras el proceso de normalización se convertía en la siguiente expresión de partida e_0:

```
letrec
  diez   = 10
  diez0  = diez@{obs1}
  diez00 = diez0@{obs2}
in diez00
```

Pasaremos ahora a ver la evaluación de dicha expresión en la máquina *Mark-2*$^{@}$. Al igual que en la semántica, en los casos en que alguno de los componentes de la configuración no varíe, éstos se mostrarán con puntos suspensivos. Por otro lado, tal y como se hizo en la semántica, sólo resaltaremos los cambios que se produzcan en cualquiera de los componentes, mientras que el resto se mostrarán también con puntos suspensivos. Nótese que en este caso no se han podado los entornos (véase Sección 2.2.4). De esta forma la clausura p_1 almacena todo el entorno, a pesar de que está claro que dicha clausura no necesita dicho entorno, ya que se trata de un entero y, por tanto, se podría haber eliminado.

	Heap	Control	Entorno	Pila	Efecto lateral	regla
	$\{\}$	e_0	$\{\}$	$[\,]$	$\{\}$	*letrec*
$\Longrightarrow$	$\left\{\begin{array}{l} p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_2 \mapsto (diez^{@\text{obs1}}, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_3 \mapsto (diezO^{@\text{obs2}}, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \end{array}\right\}$	$diezOO$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\,]$	$\{\}$	*var1*
$\Longrightarrow$	$\left\{\begin{array}{l} p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_2 \mapsto (diez^{@\text{obs1}}, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \end{array}\right\}$	$diezO^{@\text{obs2}}$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\#p_3]$	$\{\}$	*var@S*
$\Longrightarrow$	$\{\ldots\}$	$diezO^{@(0,0)}$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\#p_3]$	$\{0\ 0\ \mathit{Observe}\ \text{obs2}\}$	*var1@*
$\Longrightarrow$	$\{\ldots\}$	$diezO$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$@(0,0) : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Enter} \end{array}\right\}$	*var1*
$\Longrightarrow$	$\left\{p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\})\right\}$	$diez^{@\text{obs1}}$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$\#p_2 : [\ldots]$	$\{\ldots\}$	*var@S*
$\Longrightarrow$	$\{\ldots\}$	$diez^{@(2,0)}$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\ldots]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Observe}\ \text{obs1} \end{array}\right\}$	*var1@*
$\Longrightarrow$	$\{\ldots\}$	$diez$	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$@(2,0) : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 2\ 0\ \mathit{Enter} \end{array}\right\}$	*var1*
$\Longrightarrow$	$\{\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$\#p_1 : [\ldots]$	$\{\ldots\}$	*var3*
$\Longrightarrow$	$\left\{p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\})\right\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$@(2,0) : [\ldots]$	$\{\ldots\}$	*var3@*
$\Longrightarrow$	$\{\ldots\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$\#p_2 : [\ldots]$	$\left\{\begin{array}{l} \ldots \\ 2\ 1\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	*var3*
$\Longrightarrow$	$\left\{\begin{array}{l} p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_2 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \end{array}\right\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$@(0,0) : [\ldots]$	$\{\ldots\}$	*var3@*
$\Longrightarrow$	$\{\ldots\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\#p_3]$	$\left\{\begin{array}{l} \ldots \\ 0\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	*var3*
$\Longrightarrow$	$\left\{\begin{array}{l} p_1 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_2 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \\ p_3 \mapsto (10, \left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}) \end{array}\right\}$	10	$\left\{\begin{array}{l} diez \mapsto p_1 \\ diezO \mapsto p_2 \\ diezOO \mapsto p_3 \end{array}\right\}$	$[\,]$	$\left\{\begin{array}{l} 0\ 0\ \mathit{Observe}\ \text{obs2} \\ 0\ 0\ \mathit{Enter} \\ 0\ 0\ \mathit{Observe}\ \text{obs1} \\ 2\ 0\ \mathit{Enter} \\ 2\ 0\ \mathit{Cons}\ 0\ 10 \\ 0\ 0\ \mathit{Cons}\ 0\ 10 \end{array}\right\}$	

Como se puede observar, en la reducción de esta expresión tan simple se ven involucradas varias reglas que hacen bastante tedioso y complicado de presentar la reducción completa. Por otro lado, es interesante observar que la pila contiene una *especie de continuaciones*, de tal

forma que cuando en la expresión de control alcanzamos una forma normal la consultamos para averiguar qué hay que hacer con dicha forma normal. Al ser los enteros una especie de constructores, su comportamiento es similar al de los otros constructores.

Como cabe esperar, el fichero que se produce en la máquina abstracta y el que se produce en la reducción semántica es exactamente el mismo. □

Capítulo 7

Extendiendo Hood al entorno Paralelo

Como ya hemos visto en el Capítulo 4, en programación funcional existen varios lenguajes de programación paralela. Uno de los problemas no tratado en este ámbito es la depuración de ellos. De hecho, en la actualidad no existe ningún depurador en programación funcional paralela perezosa. Por eso, en este capítulo extenderemos la librería Hood al entorno paralelo. Más concretamente, extenderemos Hood para que funcione con los lenguajes GpH y Eden.

Una de las ventajas de que tanto GpH como Eden estén basados en Haskell y se compilen a través del compilador GHC, es que sólo es necesario realizar una única extensión de la librería Hood para que trabaje en ambos entornos paralelos. En este capítulo presentaremos los detalles más relevantes de dicha extensión, así como una aplicación para el análisis de especulación en Eden que se basa en el uso de dicha librería.

Dicha implementación se basa en la utilización de ficheros para el almacenamiento de las observaciones. Estos ficheros se considerarán regiones críticas, ya que sobre ellos se pueden querer realizar varias anotaciones a la vez y, en consecuencia, será necesario protegerlos con semáforos. Como los ficheros son manejados por el sistema operativo, se creará en cada computador un fichero de observaciones protegido con un semáforo de dicho sistema operativo. Es un poco más ineficiente que sobre otra estructura, pero como partimos de la base de que la depuración sólo se utiliza en las etapas de desarrollo de un programa, no es una ineficiencia preocupante. Además, gracias a ello se obtiene la ventaja de que obtenemos las observaciones incluso aunque el programa no finalice, con lo que se hace posible analizar los motivos por los cuales no ha finalizado.

Normalmente, en el desarrollo de una aplicación en paralelo, inicialmente se suelen realizar las pruebas en una única máquina simulando el comportamiento de n máquinas. La librería que presentaremos en este capítulo funciona tanto en una única máquina simulando la evolución paralela, como en varias máquinas ejecutándose en paralelo. La diferencia consiste en que en vez de crear un único fichero cuando estamos simulando el paralelismo en una única máquina, ya que se genera un fichero por cada sistema operativo, se crea un fichero de observaciones en cada máquina en la que estamos ejecutando el programa en paralelo.

Anteriormente se ha comentado que uno de los problemas que se puede resolver con esta librería consiste en la especulación que se produce en el lenguaje Eden cuando se crean los procesos. En este capítulo no sólo estamos interesados en analizar la especulación final del sistema, sino en analizar la especulación en cualquier momento del cómputo. Esto resulta interesante porque bajas tasas de especulación en las fases intermedias pueden provocar que algunos procesos queden bloqueados, mientras que altas tasas pueden dificultar la obtención de un reparto de carga

equilibrado. Para poder analizar la especulación en cualquier momento, será necesario realizar una leve modificación de las observaciones incluyendo el tiempo en ellas, de tal forma que sea posible compararlas temporalmente y posteriormente postprocesar dichas observaciones.

Los detalles técnicos de ambas extensiones, junto con un estudio de demora debido al uso de dicho análisis, se presentarán al final del capítulo, ya que no son necesarios ni para comprender el funcionamiento de la librería, ni las observaciones en paralelo, ni el análisis de especulación. Por este motivo, en este capítulo primero explicaremos el comportamiento de la librería, las observaciones que podemos obtener con ella y la forma de utilizarla para hacer un análisis de especulación.

El trabajo expuesto en este capítulo ha dado lugar a la publicación [ERR07].

7.1. Generando observaciones en paralelo

Para generar observaciones en paralelo ha sido necesario modificar la librería original de Hood, ya que esta librería no funcionaba correctamente en un entorno paralelo. Los detalles técnicos de la modificación se encuentran en la Sección 7.4.1. Al realizar la modificación de la librería hemos decidido mantener su comportamiento. El uso de la librería es exactamente el mismo que con la librería original, es decir, es necesario anotar los datos a observar con la función `observe` tal y como se vio en la Sección 3.4 y es necesario ejecutar el programa aplicando la función `runO` a todo el programa para que se produzcan y muestren las observaciones. De esta manera cualquier programador habituado a utilizar dicha librería podrá utilizar la librería modificada sin tener que acostumbrarse a una nueva forma de uso. Ahora bien, en Eden y GpH el uso de la función `runO` es innecesario si sólo queremos que se produzcan las observaciones sobre el fichero de observaciones, sin que éstas se muestren. Esto es debido a que como ambos lenguajes de programación funcionan en paralelo y cuando se están ejecutando en paralelos sobre diferentes sistemas operativos generan diferentes ficheros, tras la finalización será necesario recopilar y mezclar los ficheros. Por lo tanto, no tiene ningún sentido tratar únicamente el fichero del programa principal, que es lo que muestra la función `runO`, salvo en el caso de que estemos simulando el paralelismo en un único procesador.

Esto nos lleva a cierta incertidumbre con respecto a las anotaciones de observación sobre estructuras paralelas. Ya que, por un lado su comportamiento viene vinculado por el comportamiento secuencial que éstas poseían y, por otro, no sabemos en qué grado se modifica por culpa del paralelismo. Es más, esta incertidumbre nos lleva a plantearnos algunas cuestiones:

1. ¿Se pueden observar las estructuras de datos o funciones?

2. ¿Cómo se transmiten las marcas de observaciones en GpH, es decir, las funciones observadas y los constructores?

3. ¿Cómo se transmiten las marcas de observaciones en Eden, es decir, las funciones observadas y los constructores?

4. ¿Es posible observar en qué procesador se han producido los cómputos?

A la primera pregunta se puede responder para ambos lenguajes indicando que debido a que no se ha modificado el comportamiento básico de la librería se siguen pudiendo observar tanto estructuras de datos como funciones en ambos lenguajes de programación.

La respuesta a la segunda pregunta es bastante sencilla, pues como se comentó previamente GpH posee un *heap virtual* común para todas las hebras activas, es decir, cada vez que una hebra necesita un dato lo coge del *heap* y bloquea esa hebra. Por tanto, no existe ningún problema con las observaciones. Si estas hebras se evalúan en un procesador distinto del procesador del padre producirán sus observaciones en un fichero de observaciones diferente al del padre y tras finalizar el cómputo será necesario juntar ambos ficheros para obtener la observación final. Como el campo *portId* corresponde con un entero, la mezcla de los ficheros habrá que realizarla con cuidado para que se no se mezclen las observaciones. Esta tarea se puede realizar sencillamente cambiando en uno de los ficheros todas las referencias a dicho campo, por ejemplo, añadiéndolas el máximo valor obtenido en el otro fichero.

La respuesta de la tercera pregunta es un poco más complicada que la de la segunda pregunta, ya que Eden, a diferencia de GpH, posee varios *heaps*, uno por cada proceso. En este punto debemos recordar el modo de funcionamiento de Eden:

- Cuando se envía un dato a un proceso a través de un canal, se obliga a reducir dicho dato a forma normal. Debemos recalcar que en la implementación de Eden se considera que las funciones siempre se encuentran en forma normal. Por ejemplo, la función `f = case x of 3 -> \y.y+1` no se encuentra en forma normal a nivel semántico, pero a nivel de implementación sí, es decir, semánticamente la función anterior no se encuentra en forma normal porque posee una expresión **case** en cabeza. Recalcaremos, que la función que en la implementación reduce a forma normal una expresión funcional es simplemente $rnf\ f = f$.

- Cuando se crea un nuevo proceso, se copian todas las clausuras que necesite para evaluar la abstracción de proceso. Dichas clausuras se copian en el estado de evaluación que se encuentren. Nótese que esto puede llevar a repetir un mismo cómputo en más de un proceso.

Pasaremos ahora a explicar las posibilidades que se nos plantean en el tratamiento de las observaciones:

1. Que la observación corresponda a una estructura de datos que se ha de pasar como dato de entrada del proceso a través de un canal. Entonces, al reducirse dicha clausura a forma normal, que corresponde con la evaluación secuencial de dicha clausura, producirá las observaciones en el proceso padre, que es donde se hace la reducción.

2. Que la observación corresponda con una función que se pasa como dato de entrada al proceso a través de un canal. En este caso se nos plantean dos opciones:

   - Que dicha observación no haya sido reducida aún en el padre. En este caso, como las clausuras se encuentran en forma normal, se transmiten directamente al hijo sin reducirse. Por tanto, las observaciones se producirán en el proceso hijo.

     Para clarificarlo veamos el siguiente ejemplo:

```
f :: Int -> Int
f x = x + 5

g :: (Num a, Tran a, Tran b) => (a -> b) -> b
g f = f 4
```

```
fObs = observe "fun" f

p = process g

main = run0 $ print $ p # fObs
```

El proceso `p` recibe como argumento la función `fObs` que corresponde con la observación de la función `f`. `fObs` aún no ha sido demandada cuando se crea el proceso hijo `p`, por tanto, será en el proceso hijo en el que se producirá la observación, que corresponderá con la siguiente observación:

```
-- fun
   { \ 4 -> 9 }
```

- Que dicha observación ya haya sido reducida en el padre. En este caso las clausuras también se encuentran en forma normal y se transmiten directamente al hijo sin volver a reducirse. Por tanto, cuando el proceso hijo vuelva a aplicar dicha función se observará en el proceso hijo sólo la aplicación y no el *string* inicial de la observación.

  Consideremos el ejemplo anterior pero modificaremos un poco dicho ejemplo para que antes de crearse el proceso se reduzca la observación de la función, por ejemplo:

  ```
  main = run0 $ print $ case fObs 3 of
                          8 -> p # fObs
  ```

  En este caso, antes de que se cree el proceso hijo se reduce la observación de la función y la observación que se produce en el hijo corresponde únicamente con la observación de la aplicación de la función al valor 4, es decir, se produce la siguiente observación:

  ```
  -- fun
     { \ 3 -> 8
       \ 4 -> 9 }
  ```

  donde la observación `-- fun {\ 3 -> 8}` se ha producido en el padre, pero sin embargo la observación `\ 4 -> 9` se ha producido en el proceso hijo.

3. Que la observación corresponda con una clausura necesaria para la creación del proceso. En este caso dicha clausura se envía directamente sin evaluar y es el hijo el que la evalúa en caso de que la necesite a lo largo de su cómputo. Por tanto, es el hijo el que la observa. En este caso, al igual que el caso anterior, sólo se observará la reducción que se produzca en el hijo. De tal forma que el *string* inicial sólo se mostrará en el hijo si el padre no lo evaluó con anterioridad a que se creara el proceso hijo.

Este mismo análisis se puede realizar en la otra dirección, es decir, en los datos que el proceso hijo envía al proceso padre.

En el Capítulo 8 nos plantearemos diferentes opciones semánticas en el envío de las marcas de observación a los procesos. Estudiaremos la semántica de esta implementación y propondremos nuevas alternativas semánticas. La ventaja de la implementación presentada en este capítulo, como ya se ha comentado, consiste en que es más modular y fácil de mantener, debido a que no es necesario realizar ninguna modificación en el compilador. Además, como ya se verá, cualquier opción semántica puede ser simulada con esta librería, añadiendo las marcas de observación necesarias.

Finalmente y contestando a la pregunta "¿Es posible observar en qué procesador se han producido los cómputos?" debemos indicar que la respuesta es simple: la respuesta es *sí*, aunque con matices. Si estamos evaluando en paralelo con varios procesadores, en cada procesador se crea su propio fichero de observaciones. En dichos ficheros se encuentran las observaciones relativas al cómputo realizado en dicho procesador, es decir, en este caso se observa claramente el cómputo realizado en dicho proceso. Sin embargo, si estamos evaluando en una única máquina simulando el comportamiento de varias a través de PVM, sólo se crea un único fichero en el que se producen todas las anotaciones de observación. Como comentamos previamente, este tipo de evaluaciones es muy normal en la etapa inicial de generación de código, cuando se están realizando las pruebas tanto para corregir el código, como para optimizarlo y realizar ciertos análisis. No obstante, podemos paliar este problema aprovechándonos de los recursos que ofrecen tanto GpH como Eden. Ambos poseen varias variables globales, entre las que destacamos las siguientes:

`noPe :: Int` Indica el número de procesos PVM/MPI que posee el sistema.

`selfPe :: Int` Indica el número de proceso PVM/MPI en el que se encuentra el proceso.

Utilizando estas variables globales se puede averiguar parcialmente en qué proceso PVM se produjeron las observaciones. Si se quiere observar en qué proceso PVM se está computando una observación, sólo será necesario añadir en el `String` de la marca de observación el valor de `selfPe`. Hay que aclarar que tanto en GpH como en Eden con esta anotación extra sólo obtendremos en qué procesador se redujo la observación inicial, ya que una vez que esta anotación se produzca, cada una de las clausuras a las que haga referencia se observará en el procesador en que se evalúe. Es por eso por lo que comentamos que este tipo de anotaciones sólo nos da información parcial (si el padre reduce el *string* y el hijo el resto, ese tipo de información no se obtiene) de en qué procesador virtual se han generado dichas observaciones. Pero esta información puede ser total si el programador coloca las observaciones de forma adecuada. De esta manera no es necesario modificar el compilador para que cada proceso PVM posea su propio fichero de observaciones, ya que obtenemos casi los mismos resultados de una forma más sencilla, estándar y fácil de mantener en futuras versiones del compilador.

7.2. Observando los procesos en Eden

Una vez que ya hemos visto cómo se transmiten las marcas de observación tanto en Eden como en GpH. Creemos relevante estudiar las observaciones en los procesos de Eden.

Debido a la existencia de procesos en Eden, la librería de observación nos da más juego que en GpH y nos plantea más opciones y más preguntas. Las preguntas más relevantes aún sin resolver son:

- ¿son observables los procesos?
- ¿qué observar de los procesos?
- ¿cómo observar los procesos?

En esta sección analizaremos en detalle todas las posibilidades de observación de los procesos.

La respuesta a la pregunta "¿son observables los procesos?" resulta un tanto compleja. Ya que en principio la respuesta es que *no* son observables, pues no se ha definido una instancia de la

clase `Observable` del constructor `Process`. El motivo por el que no hemos realizado dicha tarea es que no parece que haya una respuesta intuitiva a dicha pregunta. Para aclarar esta respuesta creemos conveniente analizar qué se quiere observar cuando observamos un proceso. Las opciones que se nos plantean son las siguientes:

- los datos que el padre manda al proceso hijo, a través de los canales de entrada, o
- los datos que el proceso hijo utiliza, de los enviados por el proceso padre a través del canal de entrada, para realizar sus cómputos, (es decir, los que necesita el hijo), o
- los datos que el proceso hijo manda al proceso padre, a través de los canales de salida, o
- los datos que el proceso padre utiliza, de los enviados por el proceso hijo a través de los canales de salida, para realizar sus cómputos (es decir, los que necesita el padre), o
- cualquier mezcla de los anteriores.

Cualquier opción de observación es aceptable y, por tanto, cualquier definición de la instancia de la clase `Observable` de los procesos es defendible. Por ese motivo, preferimos no desarrollar la instancia de la clase `Observable` para el constructor `Process`. Además, cualquiera de esas observaciones se pueden obtener sin realizar dicha instancia y de esta manera el programador será más consciente de lo que quiere observar en cada caso.

Recordaremos que, en su esquema más simple, los procesos en Eden son funciones que reciben como parámetro, un dato o un *stream* (lista de datos) a través de un canal y devuelven como resultado, un dato u otro *stream* a través de otro canal. También pueden tener una tupla de canales, pero para mayor claridad en los ejemplos obviaremos esta opción. Por tanto, el planteamiento inicial consiste en añadir anotaciones para observar los datos que se transmiten a través de los canales o los que se utilizan de los que se transmiten. Veamos sobre un ejemplo pequeño como realizar las diferentes opciones de observación.

7.2.1. Ejemplo de observación de un proceso

Consideremos el siguiente ejemplo donde el proceso recibe una lista de enteros y devuelve únicamente los valores primos sin considerar el primer elemento. El padre únicamente enviará los datos al hijo y cogerá los 4 primeros valores devueltos por dicho proceso.

Ejemplo 7.1 Partimos del siguiente código escrito en Eden:

```
padre :: [Int]
padre = take 4 (hijo # [3..20])

hijo :: Process [Int] [Int]
hijo = process fHijo

fHijo :: [Int] -> [Int]
fHijo xs = filter isPrime (tail xs)

main = print padre
```

Si estamos interesados en ver los datos que el padre envía al proceso hijo, aunque este no los demande, sólo será necesario añadir una observación a los parámetros de entrada del proceso hijo, es decir, el código modificado del padre sería el siguiente:

```
padre = take 4 (hijo # observe "outsToProcess" [3..20])
```

De esta manera observamos los datos que el padre envía al hijo, es decir, se produce la siguiente observación:

```
-- outsToProcess
     3 :  4 :  5 :  6 :  7 : 8 : 9 : 10 : 11 : 12 : 13 : 14 : 15 :
    16 : 17 : 18 : 19 : 20 : []
```

Sin embargo, si quisiéramos observar los datos que ha necesitado el hijo para su cómputo, es decir, todos los datos que ha enviado el padre menos el primero, deberíamos modificar el hijo y anotarlo de la siguiente forma:

```
hijo = process (\ins -> fHijo (observe "insToProcess" ins))
```

De esta manera, suponiendo que el hijo tienga tiempo para tratar todos los datos de la lista de entrada antes de que el padre finalice, obtendríamos la siguiente observación:

```
-- insToProcess
     _ :  4 :  5 :  6 :  7 : 8 : 9 : 10 : 11 : 12 : 13 : 14 : 15 :
    16 : 17 : 18 : 19 : 20 : []
```

donde puede apreciarse que aunque el padre ha enviado el primer elemento al hijo, éste no ha necesitado el valor de dicho dato para su cómputo.

De forma similar podemos observar los datos que el hijo envía a su padre, aunque éste no los utilice. Esto se realizaría modificando el hijo de la siguiente forma:

```
hijo = process (\ins -> observe "outsFromProcess" (fHijo ins))
```

De esta forma obtendríamos la siguiente observación:

```
-- outsFromProcess
    5 : 7 : 11 : 13 : 17 : 19 : []
```

Ahora bien, el padre no utiliza todos esos datos para devolver su resultado. Para observar los datos necesarios por el padre para devolver su resultado haría falta modificar el padre de la siguiente forma:

```
padre = take 4 (observe "insFromProcess" (hijo #[3..20]))
```

Con esta anotación obtendríamos la siguiente observación:

```
-- insFromProcess
    5 : 7 : 11 : 13 : _
```

que indica que el padre sólo ha necesitado 4 datos de los devueltos por el hijo. □

7.2.2. Generalizando las observaciones de los procesos

Tal y como se ha visto en el ejemplo anterior, colocando las anotaciones en los sitios adecuados, es posible observar cualquiera de los datos de entrada o salida que maneja un proceso. Combinando dichas observaciones podemos definir de la forma adecuada las observaciones de los datos de los procesos.

Nuestra idea ahora consiste en realizar de forma sistemática dichas observaciones. Para ello redefinimos los constructores básicos de Eden, de tal forma que éstos realicen las tareas necesarias de observación sin que el programador se tenga que encargar de los detalles: sólo será necesario instanciar el proceso que desea observar llamando a los operadores nuevos. Entonces se observarán

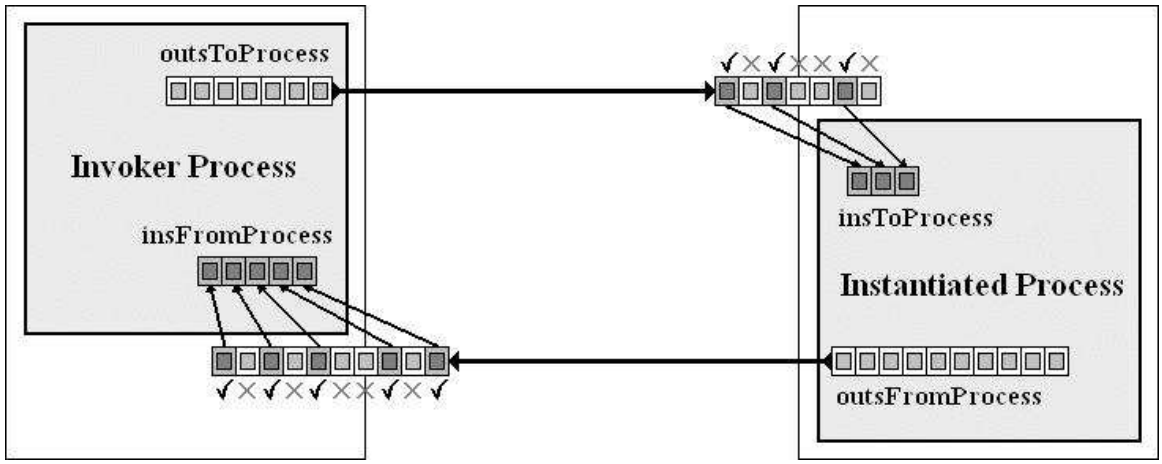


Figura 7.1: Proceso padre e hijo

los datos relativos a las entradas del proceso y a su salidas. Aplicando los nuevos constructores tanto al proceso padre como al proceso hijo, toda la información relativa al proceso se observará adecuadamente, es decir, si queremos observar los datos de entrada y salida de un proceso que computa la función `f` entonces en vez de utilizar como proceso `process f` deberemos utilizar como proceso `processObs f`:

```
processObs f = process \ins -> (observe "outsFromProcess" outs)
          where outs = f ins'
                ins' = observe "insToProcess" ins
```

La función anterior define un proceso cuyos inputs corresponden con `ins`. Para observar los datos que recibe de su padre y que él *demanda* en su cómputo sólo es necesario añadir la observación **insToProcess** a los valores recibidos por el proceso. Después de que la función `f` se aplique a sus entradas, se obtiene la salida `outs`. Para observar los datos que este proceso transmite a su creador se anota como observable sus salidas, es decir, **outsFromProcess**.

Por tanto, la función anterior nos permite observar las entradas y salidas del proceso hijo. De forma similar podemos observar el comportamiento del proceso hijo desde el punto de vista del padre. Para ello lo que hacemos es redefinir el operador de instanciación incluyendo las observaciones adecuadas. El nuevo operador `##`, basado en el operador estándar `#`, corresponde con la siguiente implementación:

```
p ## args = observe "insFromProcess"
                          (p # (observe "outsToProcess" args))
```

Este nuevo operador permite a cualquier proceso instanciar un proceso observado. Además se han añadido dos observaciones para observar las entradas y salidas del proceso hijo. Éstas corresponden con las observaciones etiquetadas con los *string*s **insFromProcess** (que muestra de los datos que recibe el padre desde el proceso hijo sólo los que necesita para realizar su cómputo) y **outsToProcess** (que muestra los datos que manda el proceso padre al hijo, sin importar si el hijo los demanda o no).

El uso de ambos constructores nos da el esquema general presentado en la Figura 7.1. Combinando las nuevas abstracciones `processObs` y `##` obtenemos los cuatro datos relevantes en un proceso. Analizando estas observaciones el lector se habrá dado cuenta que en realidad estamos observando dos funciones:

- Desde el punto de vista del padre del proceso estamos observando los datos de entrada y salida de su proceso hijo, es decir, `\` **outsToProcess** `->` **insFromProcess**.
- Desde el punto de vista del proceso hijo estamos observando los datos de entrada y salida de su función, es decir, `\` **insToProcess** `->` **outsFromProcess**

Por tanto, en el caso de que queramos observar estos elementos, las observaciones se pueden realizar anotando las correspondientes funciones. De esta manera, la definición de los constructores básicos de Eden puede simplificarse de la siguiente manera:

```
processObs f = process (observe "childProcess" f)

p ## args = (observe "parentProcess" (\x -> p # x)) args
```

Con esta modificación conseguimos la misma información de una forma más compacta, clara y sencilla. El único problema que se plantea en esta situación, y también en la anterior, es que si queremos observar los datos de varios procesos todos aparecerán etiquetados con la misma etiqueta y para comprender qué observación de los datos del proceso hijo va emparejada con qué observación de los datos del proceso padre habría que analizarlas en detalle y en muchos casos sería imposible. Por tanto, es necesario extender ambos operadores para añadir un parámetro extra: la anotación con la que el programador desea que aparezcan las observaciones. La nueva definición de los operadores es la siguiente:

```
processObs :: (Trans b, Trans a, Observable b, Observable a)
                  => String -> (a -> b) -> Process a b
processObs str f =
      process (observe  (str ++ ('_': show selfPe)) f)

instProcessObs :: (Trans b, Trans a, Observable b, Observable a)
                     => String -> Process a b -> a -> b
instProcessObs str p args =
      (observe (str ++ ('_' : show selfPe)) (\x -> p # x)) args
```

donde hemos sustituido el operador `##` por la función `instProcessObs`, debido a que ahora posee tres argumentos, y hemos añadido a la marca de observación que el programador desea `('_': show selfPe)` que da información sobre qué procesador evalúa dicha observación.

Con estas modificaciones resulta sencillo observar los datos involucrados en los procesos, pues simplemente es necesario añadir las observaciones en los lugares adecuados. Además, esta idea puede ser aplicada a otros esquemas y estructuras de programación de Eden, como por ejemplo, los esqueletos. En particular, en la siguiente sección veremos que la utilizaremos para el análisis de especulación.

7.3. Utilizando la librería para el análisis de especulación en Eden

Una de las principales motivaciones para extender la librería Hood al entorno paralelo consiste en las posibilidades que dicha extensión nos brinda para realizar diversos análisis. En particular,

el análisis en el que estamos interesados es el de *especulación*. La especulación en Eden se puede medir obteniendo la respuestas a las siguientes preguntas: ¿Cuántos datos se envían al hijo sin que este los necesite? y ¿cuántos datos devuelve el hijo que no son necesarios para el resultado final?.

Para realizar este tipo de análisis utilizaremos los dos nuevos operadores que hemos visto en la sección anterior. Aunque si bien es cierto que con esos operadores podríamos obtener la especulación final, nuestra pretensión inicial va un poco más allá y consiste en averiguar en tiempo de ejecución, en cada momento del cómputo, cuánta especulación ha habido. Por tanto, será necesario modificar adecuadamente la librería de observaciones, para que cada vez que se produzca una anotación en el fichero de observación o en los ficheros de observaciones, se almacene el instante en el que se ha producido dicha observación. De esta manera se podrá comprobar en cada momento del cómputo cuántos datos se han mandado de más, es decir, podemos observar las *carreras* entre los productores y los consumidores de datos. Los detalles técnicos de dicha modificación se encuentran en la Sección 7.4.2. Pasemos, por tanto, a la parte más relevante que consiste en la medición de la especulación.

7.3.1. Especulación en Eden

Como ya se ha comentado en la Sección 4.2, para llegar a un equilibrio entre la pereza y el paralelismo, el lenguaje Eden realiza ciertas tareas de forma impaciente. Dichas tareas incluyen el envío de datos y la demanda de resultados por parte de un proceso padre a su hijo. Gracias a esto se consigue que el paralelismo funcione apropiadamente y que se obtengan una tasas de aceleración adecuadas. Sin embargo, esto hace que los procesos en ciertas situaciones reciban y/o produzcan datos de forma especulativa.

El método que aplicaremos para la medición de la especulación se basa en utilizar la librería de observaciones vista anteriormente con las menores modificaciones posibles en el código del programa a analizar. Afortunadamente, las observaciones no provocan la evaluación de los términos bajo observación, sino que sólo observan lo evaluado. Cuando un proceso instancia otro proceso, el proceso nuevo, el hijo, demanda el cómputo de sus argumentos en el anterior, el padre. Para realizar nuestro análisis necesitamos considerar la evaluación de estos términos en dos puntos diferentes. Por un lado, las observaciones de los términos requeridas por el proceso hijo nos indican las verdaderas necesidades del proceso padre en este punto. Por otro lado, las observaciones de los términos creados por el proceso hijo nos da la información del trabajo especulativo generado por el proceso hijo. Comparando ambos valores podemos establecer la cantidad de especulación innecesaria generada en cada punto de cómputo. De hecho, no sólo obtenemos el valor final en ambos lados, sino que también obtenemos el orden en que se calcula cada parte, por tanto, podemos inferir tanto la cantidad de trabajo innecesario, como las velocidades de ambos procesos. De esta forma podemos enriquecer las capacidades de generación de estadísticas en Eden.

Como se puede intuir, las observaciones necesarias son exactamente las mismas que las que vimos en la Sección 7.2. Dependiendo de en qué datos deseemos analizar la especulación, deberemos añadir las observaciones en los sitios adecuados siguiendo las mismas técnicas que allí se vieron. Por tanto, en esta sección veremos cómo manejar adecuadamente aquellas ideas para realizar el análisis de especulación de forma sencilla.

7.3.2. Un ejemplo simple

Consideraremos un ejemplo simple pero suficientemente ilustrativo de la especulación en los datos producidos por el proceso hijo y, por tanto, enviados al proceso padre. El siguiente proceso genera la lista infinita de primos mayor o igual que un número dado `n`. El proceso recibe un número natural como entrada `n` y produce la lista `outputs` (potencialmente infinita) de los primos a partir de dicho número:

```
pprimes = process generatePrimes

generatePrimes x = if (isPrime x) then x : restOfPrimes
                                  else restOfPrimes
  where restOfPrimes = generatePrimes (x+1)
```

La lista de los primos `outputs` se obtiene al realizar una llamada a la función `generatePrimes` con el parámetro `x`. Dado un valor cualquiera para dicho parámetro `x`, esta función computa la lista de todos los primos a partir del valor `x`. Para ello genera la lista de primos a partir del valor `x+1`, es decir, `restOfPrimes` (posteriormente veremos que dependiendo de los valores demandados por otros cómputos sólo se computará una parte finita de dicha lista). El valor `x` es añadido en la cabeza de dicha lista si `x` es un un valor primo o ignorado en caso contrario.

Supongamos que estamos interesados en el uso de este proceso para obtener la primera lista de primos consecutivos mayores o iguales que un cierto número, tales que la multiplicación de sus elementos es mayor o igual que un umbral dado: `threshold`.[1] Por ejemplo, dado `initialNumber=2` y `threshold=26`, deseamos obtener la lista `[2,3,5]` pues $2 * 3 * 5 = 30$. La función `myComputation` realiza esta tarea:

```
myComputation initialNumber threshold = take neededNumber primes
  where primes   = pprimes # initialNumber
        products = scanl (*) 1 primes
        neededNumber = length (takeWhile (< threshold) products)
```

Explicaremos el comportamiento de la función `myComputation`. La ligadura `primes` representa la lista de *todos* los primos desde `initialNumber` en adelante que se computa por el proceso hijo que ejecuta la función `pprimes` con el parámetro `initialNumber`. Afortunadamente, el cómputo de `myComputation` en este caso sólo demandará una cantidad finita de elementos de la lista `primes`. Por tanto, el proceso nuevo no computará la lista infinita de primos. Ya que, cuando el cómputo de la función `myComputation` finalice, el RTS automáticamente finalizará el cómputo del proceso hijo. La ligadura `products` realiza la multiplicación de todos los elementos de `primes` utilizando la función `scanl`. La función `scanl` aplica una función a los elementos de la lista. Esto se realiza aplicando la función a cada elemento y acumulando el resultado parcial para utilizarlo para el siguiente elemento. Es necesario un acumulador inicial como parámetro. La función `scanl` devuelve una nueva lista donde los elementos son los valores acumulados tras cada elemento de la lista de entrada. En nuestro caso, `scanl (*) 1 [2,3,5]` retorna la lista `[1,2,6,30]`. La ligadura `neededNumber` computa el número de elementos de la lista `products` que son necesarios para conseguir un valor menor o igual que `threshold`. Como el elemento i-ésimo de la lista `products` nos da la multiplicación de los primeros i primos de `primes`, `neededNumber` nos da el número de elementos necesarios de la lista `primes`. El número `neededNumber` es calculado cogiendo de la

[1] Una funcionalidad similar será requerida en la Sección 7.3.4 para implementar el algoritmo `LinSolv`. En particular, para aplicar el Teorema Chino de los Restos.

lista `products` los elementos menores o iguales al umbral `threshold` y computando su longitud. Finalmente, el resultado `myComputation` se computa cogiendo los `neededNumber` elementos de la lista `primes`.

Recalcaremos que la intención del programa anterior no consiste en desarrollar de forma eficiente la mejor solución paralela, sino ilustrar las ideas básicas de nuestro método. En particular, utilizaremos el ejemplo anterior para analizar el trabajo especulativo realizado por el proceso hijo al computar la función `pprimes`. Aunque este proceso puede producir una lista infinita de números, no todos los elementos son requeridos. Sin embargo, no sabemos inicialmente cuántos elementos de la lista serán necesarios. Nosotros estamos interesados en contrastar la cantidad de primos utilizada por el proceso padre (el que realiza el cómputo de la función `myComputation`) y la cantidad de primos producida por el proceso hijo (el que computa la función `pprimes`). De hecho, todos los primos calculados por `pprimes` mayores que el último primo utilizado por `myComputation` son el resultado de haber realizado trabajo innecesario, ya que dichos primos no son necesarios para obtener la respuesta final. Recalcaremos que no es una tarea sencilla para el programador eliminar este tipo de trabajo especulativo, ya que la cantidad de trabajo especulativo realizada depende de varios factores incontrolables, entre los que se encuentra el tiempo necesario por el proceso para realizar la tarea encomendada. Ambos procesos realizarán la tarea tan rápido como puedan, pero si la velocidad de ellos no se encuentra equilibrada entonces el rendimiento del programa paralelo caerá.

A continuación mostraremos cómo puede ser modificado el programa para producir la información requerida. Dicha modificación estará basada en la introducción de las marcas de observaciones que nos mostrarán los cómputos producidos (el uso de la lista de los primos en ambos procesadores). Para observar la lista de los valores primos que se producen en el proceso hijo sólo necesitamos observar los valores de la lista de datos que envía al proceso padre sin modificarla y produciendo las observaciones en el fichero correspondiente con la anotación apropiada (**outsFromProcess**). Consecuentemente, como la introducción de observaciones es transparente con respecto a los cómputos, la información requerida se obtendría con la siguiente modificación:

```
pprimes = process (\n -> observe "outsFromProcess" (generatePrimes n))
```

Ahora bien, en Eden para minimizar el número de solicitudes entre el proceso padre e hijo, cuando el proceso hijo ha computado un nuevo valor éste se envía directamente al proceso padre. Esto significa que todos los primos computados por el proceso hijo son accesibles al proceso padre casi instantáneamente. Sin embargo, recalcaremos que Eden es perezoso dentro de cada proceso, es decir, el padre sólo utilizará los datos necesarios para realizar su cómputo. Por tanto, si en el padre marcamos como observable la lista de valores devueltos por el hijo, sólo observaremos aquellos valores necesarios para computar su resultado. En consecuencia, será necesario anotar dichos valores con otra anotación, en este caso utilizaremos la siguiente anotación **insFromProcess**:

```
myComputation initialNumber threshold = take neededNumber primes
 where primes = observe "insFromProcess" (pprimes # initialNumber)
       products = scanl (*) 1 primes
       neededNumber = length (takeWhile (< threshold) products)
```

Tras la ejecución del programa utilizando los valores `327` y `49472453` como entradas en un entorno de dos procesadores, las observaciones que obtenemos son las siguientes:

```
-- insFromProcess
  331 : 337 : 347 : 349 : _
```

```
-- outsFromProcess
  331 : 337 : 347 : 349 : 353 : 359 : 367 : 373 : 379 : 383 : 389 :
  397 : 401 : 409 : 419 : 421 : 431 : 433 : 439 : 443 : 449 : 457 :
  461 : 463 : 467 : 479 : 487 : 491 : 499 : 503 : 509 : 521 : 523 :
  541 : 547 : 557 : 563 : 569 : 571 : 577 : 587 : 593 : 599 : 601 :
  607 : 613 : 617 : 619 : 631 : 641 : 643 : 647 : 653 : 659 : 661 :
  673 : 677 : 683 : 691 : 701 : 709 : _
```

Es decir, `outsFromProcess` tiene 61 elementos, mientras que `insFromProcess` sólo tiene 4. Por tanto la cantidad de primos innecesarios computados por nuestro proceso hijo corresponde con 57 valores.

Recalcaremos que estas anotaciones son las mismas que las que se crearon para observar los datos de un proceso. Por consiguiente, para desarrollar el método general parece más adecuado utilizar los operadores generales de observación que se definieron en la Sección 7.2.2.

7.3.3. Esquema general

En el ejemplo anterior el trabajo especulativo era realizado por el proceso hijo, ya que el padre únicamente enviaba un único dato. Sin embargo, esto puede suceder al reves: El proceso padre genera un proceso hijo y posteriormente empieza a enviar datos a dicho proceso que este puede necesitar o no para sus cómputos. En este caso el proceso hijo sólo utiliza de los valores que le ha enviado el proceso padre aquellos necesarios para obtener su resultado. El método a aplicar es similar al método visto para el caso anterior pero observando las entradas del proceso. Además, el método desarrollado en el ejemplo anterior puede ser generalizado para trabajar con escenarios diferentes donde el trabajo especulativo de algún proceso ha de ser analizado. Por tanto, la forma general de realizarlo es redefiniendo los constructores básicos de Eden, de tal forma que éstos realicen las tareas necesarias de observación, sin que el programador se tenga que encargar de los detalles correspondientes a éstas, es decir, para observar la especulación que se puede producir en ambas direcciones, lo único necesario consiste en utilizar los operadores vistos en la Sección 7.2.2. De esta forma, el sistema almacena los datos relativos a las entradas del proceso y a su salidas. Aplicando los nuevos constructores tanto al proceso padre como al proceso hijo toda la información se almacena adecuadamente, es decir, si queremos observar los datos de entrada y salida de un proceso que computa la función `f` entonces en vez de utilizar como proceso `process f` deberemos utilizar como proceso `processObs "childProcess" f` y si queremos observar los datos que el padre envía y recibe del proceso en vez de utilizar `f # x` deberemos utilizar como instanciación `instProcessObs "parentProcess" f x`. Así, se obtienen todas las observaciones realizadas en la comunicación del proceso padre con su hijo, así como los datos realmente demandados. En la Figura 7.1 se marcan con ✓ los valores utilizados y con X los valores no utilizados.

Ejemplo

Para el ejemplo anterior las modificaciones necesarias son las siguientes:

```
pprimes = processObs  "childProcess" generatePrimes

myComputation initialNumber threshold = take neededNumber primes
 where
   primes = instProcessObs "parentProcess"  pprimes  initialNumber
   products = scanl (*) 1 primes
   neededNumber = length (takeWhile (< threshold) products)
```

En este caso, tras la ejecución, las observaciones que obtendríamos corresponden con las siguientes:

```
-- parentProcess
  { \ 327 -> 331 : 337 : 347 : 349 : _ }
-- childProcess
  { \ 327 -> 331 : 337 : 347 : 349 : 353 : 359 : 367 : 373 : 379 :
             383 : 389 : 397 : 401 : 409 : 419 : 421 : 431 : 433 :
             439 : 443 : 449 : 457 : 461 : 463 : 467 : 479 : 487 :
             491 : 499 : 503 : 509 : 521 : 523 : 541 : 547 : 557 :
             563 : 569 : 571 : 577 : 587 : 593 : 599 : 601 : 607 :
             613 : 617 : 619 : 631 : 641 : 643 : 647 : 653 : 659 :
             661 : 673 : 677 : 683 : 691 : 701 : 709 : _
```

Como puede apreciarse, en los datos de entrada de este proceso no se ha especulado. El único dato enviado al hijo, es decir, el 327, ha sido utilizado por él para producir su resultado.

El uso de ambos constructores nos da el esquema general para medir la especulación. Los datos que nos dan los nuevos operadores `processObs` e `instProcessObs` son los necesarios para medir la especulación en las dos direcciones. La diferencia entre los datos de entrada de la observación del proceso padre y los datos de entrada de la observación del proceso hijo nos da una medida de la especulación del proceso padre al crear el proceso hijo. Por su parte, la diferencia entre la observación de las salidas del proceso hijo y el padre nos da una medida de cuánto trabajo especulativo ha realizado el proceso hijo. Obviamente estos operadores sólo generan el o los ficheros de observaciones, posteriormente hay que analizarlos para obtener la especulación. Este tipo de análisis es trivial: para cada momento temporal hay que analizar las marcas del fichero que se han producido antes de dicho tiempo. Hay que comentar de nuevo que debido a que generamos las anotaciones de observación sobre un fichero justo en el momento en que se producen, estas observaciones se crean aunque la aplicación no finalice. Además, es posible obtener estadísticas de especulación de forma dinámica, sobre una aplicación que se esté ejecutando.

Estas ideas para la medición de la especulación pueden ser aplicadas a otros esquemas y estructuras de datos en Eden. En particular, todos los esqueletos de la librería de Eden se pueden volver a escribir de forma trivial en términos de la nueva abstracción de procesos y la nueva instanciación de procesos, que ahora realizan observaciones. De esta forma será sencillo analizar el trabajo especulativo.

7.3.4. Un caso de estudio: LinSolv

Pasemos, por tanto, a aplicar estas nuevas ideas sobre un caso de estudio lo suficientemente grande para que se pueda apreciar de forma clara el interés de un análisis de la especulación.

El algoritmo `linSolv` encuentra la solución exacta a un sistema de ecuaciones lineales de la forma $Ax = b$ donde $A \in \mathbb{Z}^{n \times n}, x, b \in \mathbb{Z}^n, n \in \mathbb{N}$. En contraste con la gran mayoría de algoritmos que habitualmente encuentran una solución aproximada usando números reales con un cierto grado de error, el algoritmo presentado aquí encuentra la solución exacta y trabaja con enteros de precisión ilimitada.

Para encontrar la solución exacta de un sistema de ecuaciones, `linSolv` utiliza el algoritmo basado en *múltiples imágenes homomórficas* [Lau82]. Ésta es una aproximación algebraica computacional y consiste en las siguientes etapas:

1. Enviar parte de los datos de entrada a las diferentes imágenes homomórficas.

2. Computar la solución en cada una de dichas imágenes.

3. Combinar los resultados de todas las imágenes para generar el resultado en el dominio original.

Esta estructura es particularmente útil para realizar las operaciones sobre enteros de precisión arbitraria. En este caso el dominio original corresponde con $\mathbb{Z}$, el conjunto de todos los valores enteros, y las imágenes homomórficas son $\mathbb{Z}$ módulo p, que denotaremos por $\mathbb{Z}_p$, siendo p un número primo. Si los números de entrada son muy grandes y cada número primo los divide en una palabra-máquina, la aritmética en las imágenes homomórficas es *barata*, ya que se puede utilizar la aritmética de precisión fija. Únicamente en la etapa de combinación de los resultados, cuando se aplica el plegado basado en el Teorema Chino de los Restos(CRA) (véase [Lip71]), la aritmética es cara, ya que se necesita la aritmética de precisión arbitraria para construir los valores resultantes. Los detalles acerca de la implementación en Haskell del algoritmo `linSolv` pueden encontrarse en [LRS+03]. Como resumen comentaremos que la parte principal a ser paralelizada consiste en solucionar cada una de las imágenes paralelas, cuya definición es la siguiente `xList = map get_homSol primes` donde `primes` es una lista infinita de números primos y `get_homSol` soluciona el sistema módulo un primo dado. Por tanto, la estructura paralela básica del algoritmo consiste en realizar todos los cómputos de las imágenes homomórficas en paralelo. Se utiliza la descomposición LU seguida de una sustitución hacia delante y hacia atrás para computar la solución `pmx` en la imagen homomórfica [PTVF92]. La principal dificultad en esta paralelización consiste en que algunos primos resultan ser "desafortunados", es decir, el determinante de la matriz inicial A en la imagen homomórfica generada por el primo es cero. Cuando el primo es desafortunado no podemos utilizarlo para el resultado final. Por dicho motivo, no es posible saber a priori cuántos primos necesitaremos para resolver el sistema, aunque sí que es posible conocer una cota inferior.

Recalcaremos que estos cálculos son similares a los que se producían en el ejemplo simple que mostramos en la sección anterior. Sin embargo, en este caso la tarea de los procesos no sólo consiste en generar los primos, sino que además tienen que solucionar el sistema lineal módulo dicho primo.

Primera versión

Pasaremos ahora a explicar la primera versión paralela de dicho algoritmo y a analizar la especulación que se produce en dicha versión.

Aunque la situación ahora parece más compleja que en el ejemplo presentado en la sección anterior, nuestra estrategia para comprobar cuánto trabajo necesario se ha realizado es la misma. Ya que tenemos que solucionar un sistema lineal módulo varios números primos, el esquema paralelo más obvio consiste en utilizar el esqueleto `map_par` tal y como se vio en la Sección 4.2, de esta forma se crea un proceso independiente por cada valor primo. Para ello será necesario añadir las observaciones al esqueleto. El esqueleto modificado se presenta a continuación:

```
map_parObs f xs = [ instProcessObs ("procesoPadre"++show i)
                          (processObs ("procesoHijo"++show i) f)
                            x | (x,i) <- zip xs [1..]] `using` spine
```

```
xList_all = map_farm get_homSol primes
xList = filter lucky xList_all
```

Figura 7.2: `linSolv` paralelo en Eden (versión especulativa).

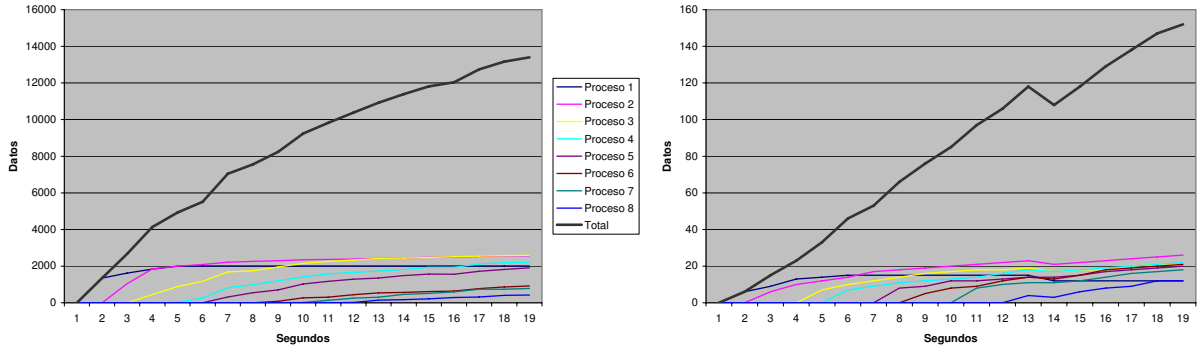


Figura 7.3: `linSolv`. Granja con 8 procesos y 9 procesadores (versión especulativa). Matriz de 10*10. Entradas y salidas.

Como puede observarse, la modificación consiste en observar cada hijo y cada solución obtenida en el padre. Lógicamente, es necesario identificar adecuadamente cada proceso. Por ese motivo se ha añadido en la anotación el entero `i`. Sin embargo, tal y como se comentó en la Sección 4.2, es mejor utilizar el `map_farm` para evitar la creación de muchos procesos. Recordemos que este esqueleto realizaba una llamada al esqueleto `map_par` que ahora será necesario modificar por una llamada al esqueleto `map_parObs`. De esta manera obtendremos las observaciones de especulación, como queda patente, sin apenas modificaciones en el código. Así pues, en nuestra primera aproximación (mostrada en la Figura 7.2) nosotros reemplazaremos la función del nivel superior `map` por el esqueleto paralelo `map_farm`.

A continuación pasaremos a estudiar la especulación que se produce en esta aproximación. Para ello aplicaremos dicho algoritmo sobre una matriz de 10*10 con 9 procesadores (`np = 9`), es decir, ocho procesos hijos y un proceso padre que se encarga de distribuir las tareas. En la Figura 7.3 izquierda se observa que la especulación del padre en el envío de datos a lo largo del tiempo a cada hijo alcanza cotas demasiado elevadas. Para representar esta gráfica, tras obtener todas las observaciones, se ha analizado el fichero de observaciones segundo a segundo. La especulación en los datos de entrada ha alcanzado la cota de 13402 datos enviados de forma especulativa por el padre a sus hijos. Por otra parte, en la Figura 7.3 derecha se observa la especulación que se produce en el envío de resultados de los hijos al padre. La cantidad de datos, aunque mucho menor, tampoco es despreciable: 152 datos enviados por los hijos al padre sin que hayan sido necesarios para el cálculo del resultado final. El problema radica en que estamos considerando una lista infinita de primos y estamos enviando indiscriminadamente dichos datos a los procesos. Por este motivo la especulación en ambas gráficas sigue una tendencia al alza y apenas desciende. Por consiguiente, en este caso dicha especulación genera un efecto negativo sobre el cómputo. El problema es mayor a medida que consideramos problemas de entrada más grandes, pues hay más tiempo durante el cual se especula sin control. Por ejemplo, consideremos

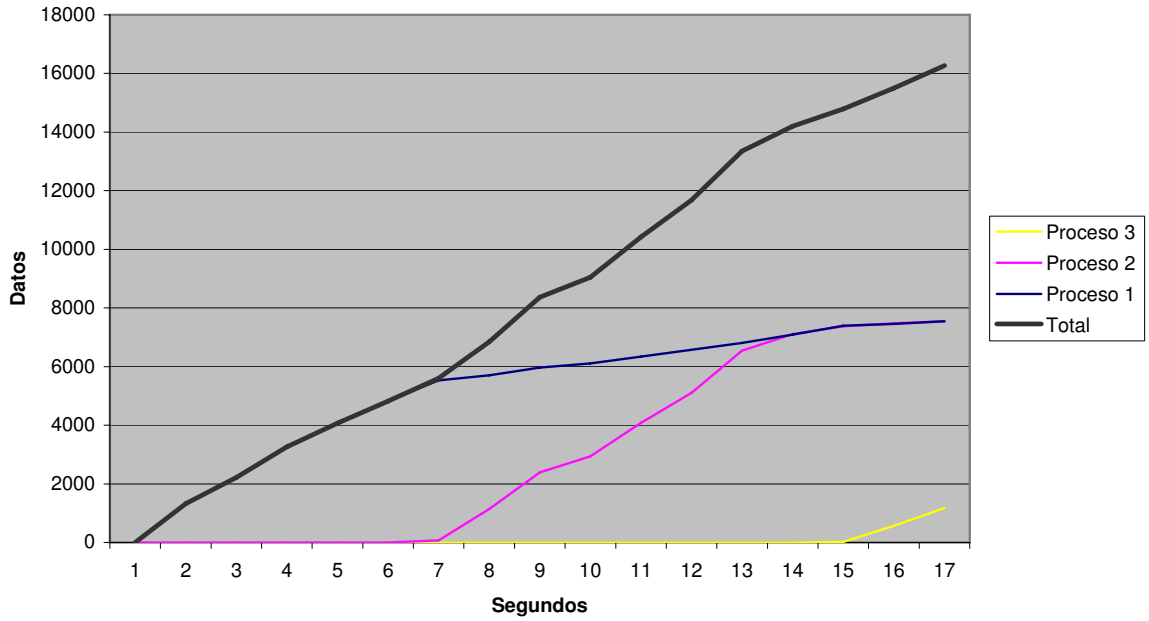


Figura 7.4: `linSolv`. Granja con 8 procesos y 9 procesadores (versión especulativa). Entradas.

una matriz densa de 20*20 y la misma configuración que en el caso anterior con respecto a los procesos y los procesadores. Esta será la matriz y la configuración con la que realizaremos todas las pruebas de las distintas versiones. En la Figura 7.4 se observa la especulación del padre en el envío de datos a los hijos a lo largo del tiempo. En este caso, al igual que en el caso anterior, la especulación almacenada transcurridos los 17 primeros segundos alcanza cotas demasiado elevadas, 16271 valores enviados a los tres hijos sin que éstos tengan tiempo de procesarlos y sin que dichos datos tengan la más mínima utilidad para el resultado final. De hecho, la especulación final es unas 6 veces superior. Al igual que en el caso anterior, el problema surge debido a que la lista de primos que estamos enviando a los hijos para que resuelvan el sistema módulo dichos primos es infinita.

Segunda versión

Como se ha podido apreciar, la primera versión genera mucha especulación debido a que la lista de primos que estábamos calculando era infinita. La solución clara a dicho problema consiste en limitar la lista de primos utilizando una versión conservadora. En este tipo de problema es sencillo calcular una cota inferior de primos necesarios para resolver el sistema de ecuaciones módulo primos. Por tanto, generaremos la lista de primos mínimos necesarios para resolver el sistema. El código puede observarse en la Figura 7.5, donde los primos se han dividido en dos listas: `p_needed` que contiene los mínimos valores primos necesarios; `p_spec` contiene el resto de primos por si alguno de los primos anteriores resulta desafortunado. La función `additional` añade un nuevo valor a la lista de tareas `primes'` cada vez que un primo resulta fallido. Las consecuencias de esta aproximación son las siguientes:

```
xList_all = map_farm get_homSol primes'
xList = filter lucky xList_all
xList_unlucky = filter (not.lucky) xList_all

(p_needed, p_spec) = splitAt ( 1 + toInt noOfPrimer) primes
primes' = p_needed ++ (aditional xList_unlucky p_spec)

additional :: [Integer] -> [Integer] -> [Integer]
additional xs ys = zipWith (\ x y -> y) xs ys
```

Figura 7.5: `linSolv` paralelo en Eden (versión conservadora)

1. La paralelización se realiza algo más tarde, ya que es necesario calcular previamente la cota mínima de primos necesaria.

2. Los cómputos finales son casi secuenciales, ya que si se necesitan más primos éstos se van enviando y calculando bajo demanda.

3. La especulación se anulará, ya que sólo si es necesario un nuevo valor primo éste se calculará y enviará a un proceso.

4. Tanto el padre como los procesos hijos no calcularán ni enviarán datos innecesarios para resolver el sistema. De esta manera tanto los cálculos como las comunicaciones se reducirán drásticamente.

Analizaremos ahora las gráficas de especulación que se obtienen en este caso: En la Figura 7.6 se puede ver la especulación existente en los datos de entrada de los procesos:

- Cada vez que se crea un nuevo proceso hijo la especulación se incrementa, debido a los valores primos que el padre envía a cada hijo para que resuelva el sistema módulo dichos primos. Esta comunicación se realiza en el momento en que se crea el proceso. Una vez creado el proceso, la especulación comienza a descender, pues el proceso comienza a utilizar dichos valores para realizar sus cálculos.

- A partir de la creación del último proceso la especulación disminuye, ya que los procesos tienen todos los datos de entrada necesarios. Por tanto, a medida que van utilizándolos, los datos dejan de ser datos enviados de forma especulativa.

- No existe especulación final ya que sólo enviamos los valores primos necesarios para la resolución del sistema.

Por otro lado, en la Figura 7.7 se puede ver la especulación existente en los datos de salida de los procesos:

- Como puede apreciarse, la especulación crece hasta el segundo 92, momento en el que todos los procesos ya han enviado al menos un resultado. Esto se debe a que el padre espera ordenadamente a recibir un valor de cada hijo. Por tanto, hasta que no recibe datos de todos los hijos sólo ha utilizado a lo sumo uno de los datos recibidos de cada hijo.

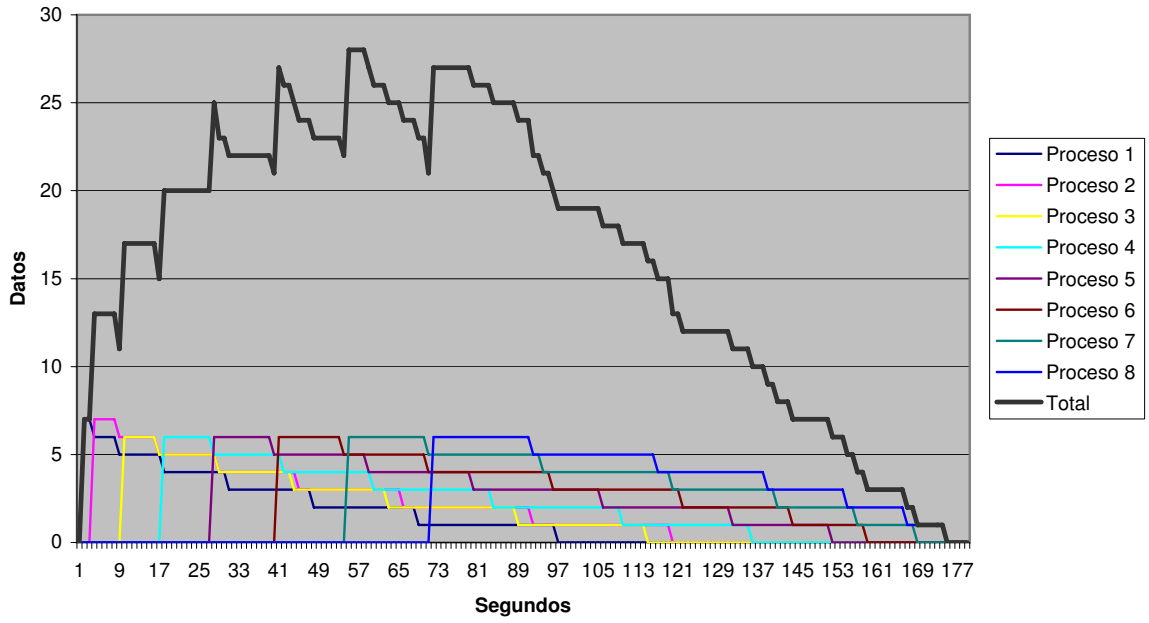


Figura 7.6: `linSolv`. Granja con 8 procesos y 9 procesadores (versión conservadora). Entradas.

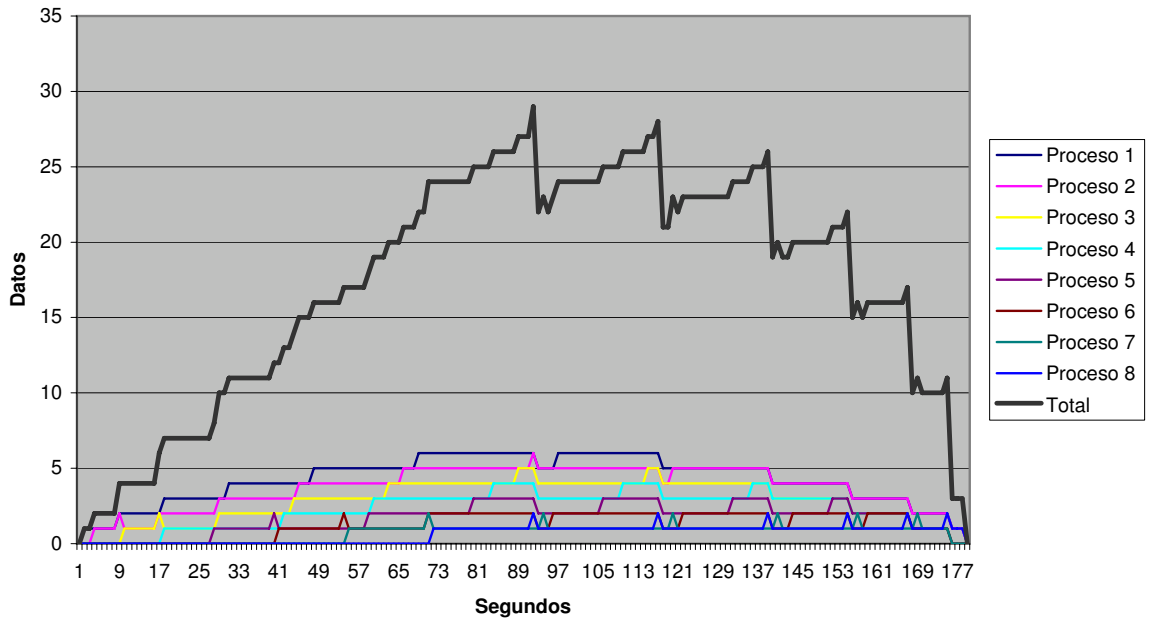


Figura 7.7: `linSolv` Granja 8 procesos y 9 procesadores (versión conservadora). Salidas.

```
xList_all = map_rw get_homSol primes'
xList = filter lucky xList_all
xList_unlucky = filter (not.lucky) xList_all

(p_needed, p_spec) = splitAt ( 1 + toInt noOfPrimer) primes
primes' = p_needed ++ (aditional xList_unlucky p_spec)

additional :: [Integer] -> [Integer] -> [Integer]
additional xs ys = zipWith (\ x y -> y) xs ys
```

Figura 7.8: `linSolv` paralelo en Eden (versión conservadora con trabajadores replicados)

- La especulación sigue una tendencia a la baja una vez que todos los procesos ya se encuentran produciendo datos, pues el padre tiene que ordenar los datos resultantes. Por ese motivo la especulación a partir de ese momento fluctúa: crece hasta que el hijo más lento manda un resultado, y una vez que se recibe el dato de este hijo, la especulación cae drásticamente. Así pues, el último hijo es el responsable de estas fluctuaciones.

- En las etapas intermedias, la especulación máxima es alcanzada por el proceso más rápido, posteriormente el segundo, tercero, etc, es decir, de forma ordenada y escalonada.

- La especulación final es nula debido a la limitación en la lista de primos, es decir, todos los datos calculados son necesarios para el resultado final obtenido.

Desde el punto de vista de la especulación final, esta implementación es óptima, pues no se realiza ningún trabajo innecesario, Ahora bien, las altas cotas de especulación intermedia podrían conducir a malos repartos de carga entre procesadores. Nótese que como ya se ha comentado previamente altas especulaciones intermedias implican que se decide demasiado pronto qué procesador se encargará de ejecutar cada tarea, de modo que se dificulta un ajuste dinámico en función de las velocidades con las que hayan realizado las tareas los procesadores. El principal problema de esta versión es que el proceso más lento puede afectar drásticamente a la eficiencia global. Así pues, sería conveniente utilizar algún otro esquema que equilibre mejor el reparto de carga entre procesos.

Tercera versión

En este caso pretendemos resolver el problema, de tal forma que se optimice el aprovechamiento de los recursos y realice un reparto de carga más justo. Para ello será necesario utilizar el esqueleto denominado "trabajadores replicados" [KPR00].

Como se comentó en la Sección 4.2, cada trabajador posee un *buffer* en el que almacena las tareas por tratar, de esta manera el trabajador no tiene que esperar a que su padre le envíe un nuevo valor para que él continúe con su trabajo. Además, el padre se encarga de enviar una tarea nueva a cada hijo una vez que éste termine una de sus tareas. En la Figura 7.8 se presenta la modificación necesaria en el código para el cálculo del `linSolv`. Como puede apreciarse, el único cambio que hemos aplicado con respecto a la versión anterior consiste en utilizar el esqueleto de los "trabajadores replicados" en vez del `map_farm` sobre la lista limitada de primos. De esta manera mejoraremos el aprovechamiento de los recursos sin producir especulación.

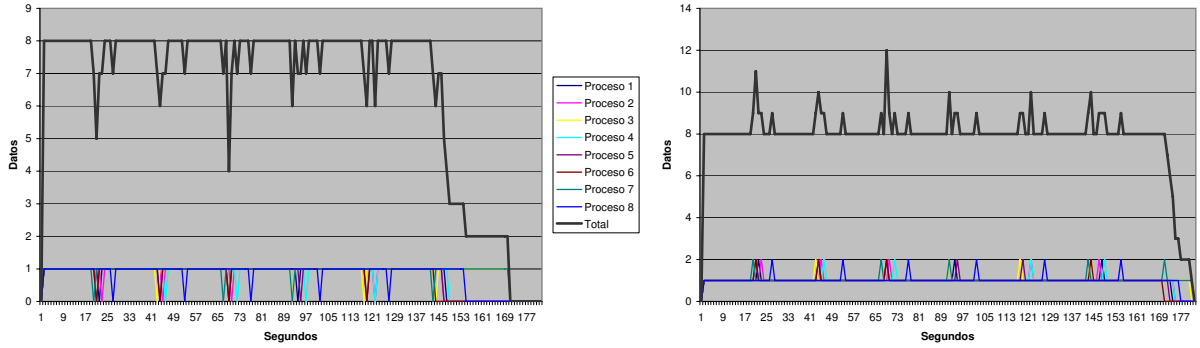


Figura 7.9: `linSolv`. "Trabajadores replicados" con 8 procesos, 9 procesadores y *buffer* de 2 datos (versión conservadora). Entradas y salidas.

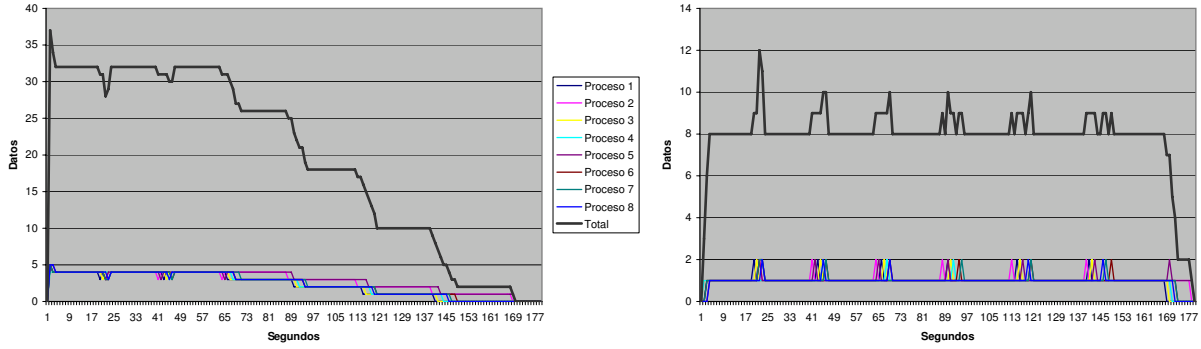


Figura 7.10: `linSolv`. "Trabajadores replicados" con 8 procesos, 9 procesadores y *buffer* de 5 datos (versión conservadora). Entradas y salidas.

Veremos y compararemos qué sucede cuando utilizamos dicho esqueleto con un *buffer* de tamaño 2 y con un *buffer* de tamaño 5. En la Figura 7.9 se presenta la gráfica con la especulación de los datos de entrada y salida con un *buffer* de 2 datos en cada hijo. Por su parte, en la Figura 7.10 se presenta la especulación en los datos de entrada y salida con un *buffer* de 5 datos en cada hijo. Como puede apreciarse, las gráficas de especulación producidas en la salida de los datos de los hijos al padre son muy similares. De hecho, durante la mayor parte del cómputo, en los datos producidos por los hijos se mantiene una tendencia de especulación cercana al valor 8, es decir, al número de procesos.

Con respecto a la especulación en los datos de entrada, cabría destacar que ambas gráficas son aparentemente muy diferentes. Si consideramos la primera mitad de ambas gráficas podemos apreciar que éstas se encuentran escaladas. La especulación máxima que se puede producir en cada una es: número de elementos del *buffer* por el número de procesos (es decir, `tamaño_buffer * número_procesos`). Por tanto, en la Figura 7.9 la especulación máxima sería 16 y en la Figura 7.10 sería 40. Sin embargo, las especulaciones máximas obtenidas en los datos de entrada son de 8 y 37 datos respectivamente. En el primer caso es la mitad y en

el segundo caso se encuentra muy proximo a la cota máxima, aunque este dato es puntual y surge en el momento de creación de los procesos: a partir de ese momento la especulación se mantiene en 32 hasta el segundo 66 en el que se produce una caída drástica debido a que el padre acaba de terminar de enviar todos los primos necesarios a sus hijos. Nótese que en la práctica el nivel de especulación habitual durante la ejecución es el mismo en los dos casos: `(tamaño_buffer -1)* número_procesos`.

La gráfica obtenida para el tamaño de *buffer* 5 parece muy diferente a la que se produce con el tamaño de *buffer* 2, aunque bien es cierto que las diferencias son sólo aparentes. Como ya se ha comentado, el comportamiento y la especulación hasta el segundo 66 es prácticamente el mismo que con el tamaño de *buffer* 2 salvo que se encuentra escalado. En los datos de entrada de los procesos se acumula una especulación cercana al 4 y la tendencia total se encuentra cerca de 32 datos especulativos. No obstante, a partir de dicho momento las cosas cambian. Al no haber más valores que enviar a los procesos, éstos comienzan a consumir los valores que tienen almacenados en el *buffer* y, por tanto, a partir de ese momento comienza una tendencia a la baja en la especulación de los datos de entrada, que en este caso se aprecia mejor que con el tamaño del *buffer* 2. Debido a que el tiempo necesario para realizar la resolución del sistema módulo un primo es parecido entre los primos que poseen los procesos hijos, ésta es escalonada. De tal forma que los escalones se deben a los momentos en que los procesos hijos comienzan a reducir nuevamente un valor de su *buffer*. En este caso podemos ver que en el segundo 145 el proceso 6 es el único que ha utilizado todos los valores de entrada, mientras que en el segundo 150 sólo quedan dos procesos por utilizar sus datos de entrada, los procesos 2 y 5. Por tanto, aunque las tareas son parecidas en cuanto al coste de tiempo, pueden apreciarse pequeñas diferencias que hacen que al final haya varios procesos que terminen más tarde, reduciéndose el paralelismo en los últimos momentos del cómputo. Como consecuencia, se incrementa parcialmente el tiempo necesario para obtener la solución final.

Analizando la especulación individual en los datos de entrada de los procesos podemos apreciar que en ambos casos los hijos utilizan uno de los datos inmediatamente. Aunque cuando el tamaño del *buffer* es 5, debido a que se envían más datos inicialmente a los hijos, puede apreciarse el momento inicial donde el hijo tiene el *buffer* lleno. Cuando el *buffer* tiene una capacidad de 2 datos existen momentos en los que los hijos se quedan con el *buffer* vacío, pero estos momentos son puntuales y puede observarse que inmediatamente el *buffer* se vuelve a llenar. Gracias a esto, los procesos nunca se quedan parados. Por su parte, cuando el tamaño del *buffer* es 5 para cada hijo nunca desciende del valor 3, salvo en los últimos momentos del cómputo. Por tanto, parece que no es necesario que los hijos posean un buffer tan grande.

En este caso la mejor aproximación se obtiene si dotamos a los hijos de una capacidad de almacenamiento de 2 datos en su *buffer*, pues, como se acaba de comentar, en la práctica es suficiente para mantener a los hijos continuamente trabajando, y reduce el número de tareas que el padre debe enviar de forma especulativa. Por tanto, se obtiene un mejor reparto de carga en los procesos.

Cuarta versión

Veamos ahora qué sucede si eliminamos el cálculo inicial de la cantidad de primos que enviamos a los procesos y utilizamos, por tanto, una lista infinita de primos. Cabe esperar que se realice trabajo especulativo, pero probablemente ganemos en eficiencia ya que el cómputo paralelo se mantendrá y no será necesario el cálculo inicial de la cantidad de primos.

```
xList_all = map_rw get_homSol primes
xList = filter lucky xList_all
```

Figura 7.11: `linSolv` paralelo en Eden (versión semi-especulativa con trabajadores replicados)

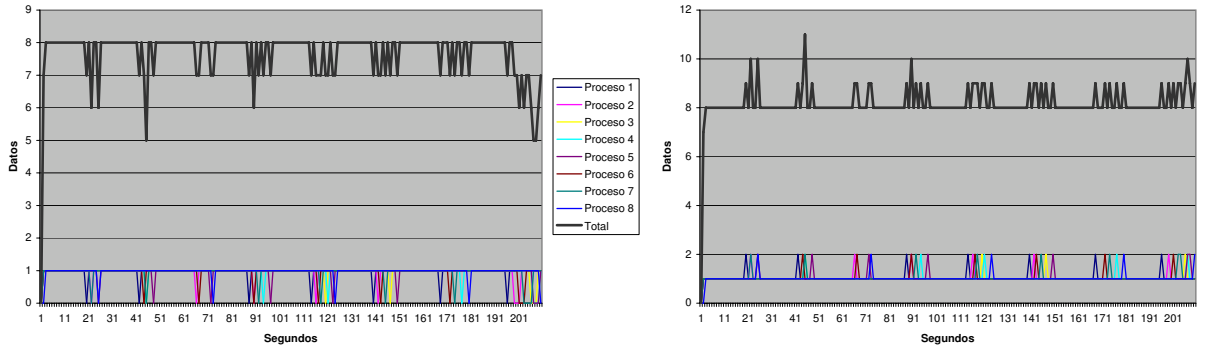


Figura 7.12: `linSolv`. "Trabajadores replicados" con 8 procesos, 9 procesadores y *buffer* de 2 datos (versión semi-especulativa). Entradas y salidas.

Las condiciones en las que trabajamos son las mismas que en la versión anterior: un proceso padre encargado de enviar una tarea nueva a cada hijo una vez que éste termine una de sus tareas y los hijos poseen un *buffer* donde almacenar las tareas pendientes de procesar. En la Figura 7.11 se presenta la modificación necesaria en el código para el cálculo del `linSolv`. Como puede apreciarse, el único cambio necesario con respecto a la versión secuencial consiste en utilizar el esqueleto de los "trabajadores replicados" en vez del `map` sobre la lista infinita de primos. Con esta versión, como veremos más adelante, controlaremos la especulación dentro de unos límites razonables.

Al igual que en el caso anterior, veremos y compararemos qué sucede cuando utilizamos dicho esqueleto con un *buffer* de tamaño 2 y con un *buffer* de tamaño 5. En la Figura 7.12 se presenta la gráfica con la especulación de los datos de entrada y salida usando un *buffer* de 2 datos en cada hijo. Por su parte, en la Figura 7.13 se presenta la especulación en los datos de entrada y salida con un buffer de 5 datos en cada hijo. Como puede apreciarse, ambas gráficas son muy similares, la principal diferencia se encuentra en los datos de entrada, como cabe esperar, y consiste en que ambas gráficas se encuentran escaladas de igual manera que en el caso anterior. La diferencia principal con respecto a la tercera versión es que ahora las gráficas son más uniformes y la especulación se mantiene estable.

Con respecto a la especulación final en los datos de salida, en ambos casos es de 8 datos, es decir, un dato especulativo por cada uno de los procesos creados. Nótese que esto no sólo representa un valor muy bajo en cuanto a especulación, sino que en la práctica es casi equivalente a no haber hecho ningún trabajo adicional: sólo se trabaja en una tarea de más mientras el proceso padre detecta que no necesita nada más. Así pues, si los procesos no estuvieran trabajando durante este tiempo en las tareas especulativas, hubieran estado ociosos y el cómputo global hubiera acabado en el mismo momento.

En este caso se puede observar de forma más clara que en la versión anterior que parece

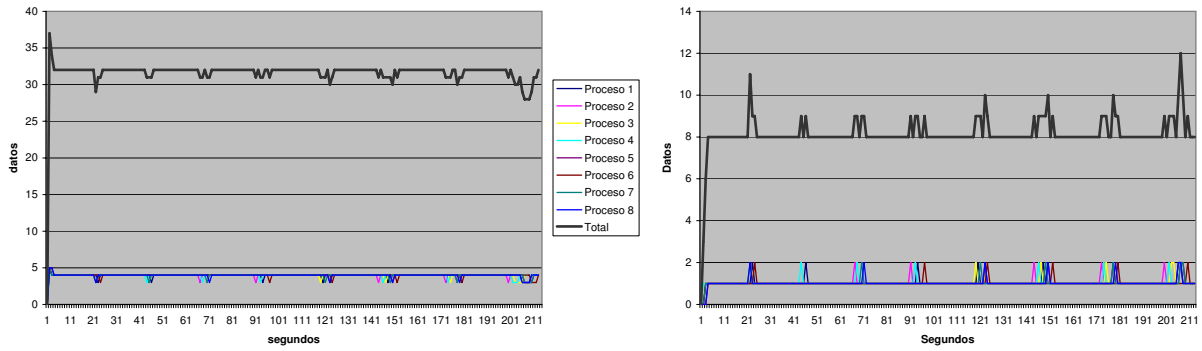


Figura 7.13: `linSolv`. "Trabajadores replicados" con 8 procesos, 9 procesadores y *buffer* de 5 datos (versión semi-especulativa). Entradas y salidas.

mejor la aproximación donde el *buffer* tiene un tamaño de 2. Ya que cuando el *buffer* tiene un tamaño de 5 los hijos nunca utilizan más de 2 datos de su *buffer*. Con un *buffer* de 2 datos existen momentos donde el *buffer* se queda vacío, pero esto no implica que el proceso se quede parado, sino que acaba de empezar a evaluar su nueva tarea. Puede apreciarse también que el padre envía un nuevo dato a dicho hijo casi instantáneamente. Gracias a que las tareas de los procesos son suficientemente costosas como para que éstos no se queden parados sin datos de entrada, se mantiene de forma más equitativa la carga de cada proceso, ya que cuando un proceso finaliza con un cálculo se le envía el siguiente valor a almacenar para su posterior procesamiento. En caso de que el tamaño del *buffer* fuera menor habría momentos en los que el proceso se quedaría parado esperando a recibir nuevos datos. Esto supondría un empeoramiento en el rendimiento y, en consecuencia, en el tiempo de ejecución.

Versión recomendada

Una vez realizado el análisis de todos los posibles casos creemos conveniente comparar los resultados y recomendar una versión. Como ya quedó patente, la primera versión paralela de este algoritmo es completamente inviable, debido a la gran cantidad de especulación que genera. Por tanto excluiremos esta versión del análisis final.

El reparto de carga en la versión segunda es peor que en la tercera y cuarta pues al especular mucho demasiado pronto impide adaptar dinámicamente el reparto de carga. En este caso no es mucho peor ya que el tiempo de cómputo necesario por cada primo es parecido, pero en general el esqueleto denominado de los "trabajadores replicados" realiza un reparto de carga mejor, pues se ajusta dinámicamente a las velocidades relativas de los procesos..

Tanto en la versión tercera como en la cuarta el tamaño del *buffer* óptimo se consigue con tamaño 2. Ya que de esta forma los trabajadores nunca se quedan esperando a recibir un nuevo dato para proseguir con sus cómputos y tampoco almacenan tareas en exceso. Almacenar tareas en exceso conlleva un empeoramiento del reparto de carga y, por tanto, del paralelismo, ya que si a un proceso le tocan las tareas más pesadas, este proceso seguirá trabajando cuando los demás procesos hayan terminado.

Las principales diferencias entre la versión tercera y cuarta son que en la tercera no se produce

especulación y la modificación del algoritmo. Como ya se ha comentado, la versión tercera no produce especulación final, pero a cambio es necesario montar una nueva estrategia de cálculo de primos. Lo primero que hay que hacer consiste en calcular el número de primos necesarios, esta tarea se realiza antes de comenzar a generar los procesos y, por tanto, reduce el paralelismo. Debido a que la cantidad de primos sólo es una cota inferior, posteriormente hay que montar una nueva estructura donde se añadan paulatinamente los primos extras que sean necesarios, es decir, desde el punto de vista del programador hay que cambiar la estrategia en la resolución del problema, pudiendo con ello cometer nuevos errores de programación. Sin embargo la versión cuarta no implica apenas tarea de modificación del código del programa. Sólo es necesario utilizar el esqueleto `map_rw` en vez de la función `map` de la versión secuencial. Aunque bien es cierto que en esta versión se produce especulación, esta especulación sólo se genera cuando los procesos quedan ociosos, es decir, al final de cómputo, por lo que no se pierde nada.

Por todos los motivos vistos anteriormente consideramos que la mejor versión consiste en la versión cuarta con un *buffer* de dos elementos. Aunque en esta versión se produzca algo de especulación, tanto la sencillez para generarla como las mejoras obtenidas nos hacen que la consideremos la mejor opción. A pesar de que el análisis detallado de tiempos de ejecución en máquinas paralelas queda fuera del alcance de esta tesis (véase [Rub01] para detalles sobre dicho tema), merece la pena comentar que la versión cuarta es, efectivamente, la que mejores aceleraciones obtiene.

Como puede apreciarse sobre este ejemplo realista, el análisis de especulación, entre otros, puede servir para decidir cómo obtener una mejor solución paralela para un problema dado. En trabajos anteriores (véase [Seg01, Rub01]) se habían definido distintos tipos de análisis para ayudar a generar programas paralelos en Eden con un buen grado de eficiencia. En este sentido, el análisis presentado en el presente capítulo debe entenderse como una ayuda más para el programador.

Finalmente, nos gustaría destacar que la generación de gráficas de especulación puede ayudar a los programadores inexpertos a entender mejor el funcionamiento real del lenguaje. Desde este punto de vista puede considerarse como un útil complemento didáctico.

7.4. Detalles de implementación de Hood en paralelo

En esta sección describiremos los detalles técnicos de implementación de nuestra extensión paralela de la librería de observación junto con las modificaciones necesarias para el análisis de especulación. Estos detalles no son necesarios ni para utilizar dicha librería, ni para comprender el comportamiento de la misma. Por ese motivo, aquellos lectores interesados únicamente en un nivel más abstracto de la definición y uso de la librería, en vez de en los detalles técnicos de cómo se ha implementado, pueden pasar directamente al siguiente capítulo, sin que con ello se pierda ninguna información relevante para ellos.

7.4.1. Extendiendo la librería Hood al entorno paralelo

La librería Hood está implementada bajo la premisa de que la única extensión necesaria para que funcione es la función `unsafePerformIO :: IO a -> a`, de tal forma que funciona en cualquier compilador de Haskell que posea dicha función no estándar. Nuestra pretensión al desarrollar dicha extensión es mantener dicha filosofía, de tal forma que no sea necesario realizar

ninguna modificación sobre el compilador. De esta manera será sencillo el uso de dicha librería en diferentes compiladores. Sin embargo, a nivel semántico nos plantearemos varias opciones, algunas de las cuales conllevarían una modificación del compilador.

El principal problema que nos encontramos cuando comenzamos a extender dicha librería al entorno paralelo es que la librería utilizaba como semáforos unas variables de tipo `MVar`. Estas variables funcionan correctamente en la versión secuencial. Pero, sin embargo, en paralelo estas variables no funcionan de manera adecuada debido a la copia de clausuras. Para comprender su comportamiento hay que considerarlas como punteros. Consecuentemente al copiarse la clausura que contiene dicho puntero entre procesadores (o en otro *heap* en el mismo procesador) dejan de tener sentido, ya que referencian una zona de memoria que en el nuevo procesador puede contener cualquier otro dato y, por tanto, generan errores en ejecución. Por este motivo nuestro planteamiento ha sido el de utilizar semáforos reales del sistema operativo. Sin embargo, Haskell no posee librerías para el uso de dichos semáforos.

Afortunadamente, Haskell posee varias opciones de unión con otros lenguajes de programación (véase HDirect[FLMP99, Fin99]). Además, es muy sencillo realizar llamadas a funciones C. Así pues, nosotros nos hemos aprovechado de las directivas de manejo de semáforos que C posee, que se encuentran en la librería *sys/sem.h*. Ya que las llamadas a funciones de C se realiza de forma simple a través de la función `_ccall_`. A continuación pasaremos a describir en detalle dichas operaciones:

`key_t ftok(char *pathname, char proy)`: Crea una llave basándose en los valores de `proy` y `pathname`. El parámetro `pathname` es el nombre de un fichero accesible al proceso que solicita la creación de la llave y `proy` es el identificador del proyecto. Para el mismo fichero y diferentes valores del segundo parámetro se crearán llaves diferentes. Si se produce algún tipo de error, bien porque no existe o bien porque no se puede acceder al fichero, la llamada devolverá el valor `-1` convertido al tipo `key_t`.

`int semget (key_t key, int nsems, int semflag)`: Crea una lista de semáforos y devuelve un identificador a dicha lista. El parámetro `key` corresponde con la llave, `nsems` es el número de semáforos que queremos crear con dicha llave y `semflag` son los permisos de acceso a dichos semáforos. Estos permisos son similares a los usados en sistemas Unix para el manejo de ficheros: de lectura y escritura para el usuario, grupo y otros.

`int semctl (int semid, int semnum, int semcld, union semun arg)`: Realiza la operación indicada por `semcld` para el semáforo número `semnum` de los semáforos identificados por `semid`. El primer semáforo al que hace referencia `semid` corresponde con el semáforo número 0. La estructura del cuarto parámetro corresponde con la siguiente:

```
union semun {
            int val;
            struct semid_ds *buf;
            unsigned short  *array;
} arg;
```

Entre las operaciones que se pueden realizar destacaremos las siguientes:

`IPC_RMID`: Borrar inmediatamente el conjunto de semáforos y sus estructuras de datos, despertando todos los procesos en espera. En este caso el argumento `semnum` es ignorado.

`SETVAL`: Poner el valor del semáforo `semnum` del conjunto de semáforos referenciados por `semid` al valor que contenga el campo `val` de `arg`. Los procesos que están durmiendo en la cola de espera son despertados si `semval` se pone a 0 o si se incrementa. El proceso que realiza la llamada ha de tener privilegios de escritura en el conjunto de semáforos.

`GETVAL`: Obtener el valor del semáforo `semnum` del conjunto de semáforos referenciados por `semid`. El proceso que realiza la llamada ha de tener privilegios de escritura en el conjunto de semáforos.

`int semop (int semid, struct sembuf *sops, size_t nsops)`: Esta función ejecuta operaciones en los miembros seleccionados del semáforo indicado por `semid`. Cada uno de los `nsops` elementos en el array apuntado por `sops` especifica una operación a ser realizada en un semáforo por `struct sembuf` que incluye los siguientes campos:

```
struct sembuf {
            short sem_num;
            short sem_op;
            short sem_flg;
} arg;
```

`sem_num`: Es el índice del array del semáforo sobre el que queremos actuar.

`sem_op`: Es el valor en el que queremos incrementar el semáforo.

`sem_flg`: Son flags que afectan a la operación.

Utilizando dichas funciones es posible crear semáforos del sistema en vez de utilizar las variables `MVar` que, como se ha comentado anteriormente, no funcionan correctamente para nuestros propósitos. De esta forma es posible bloquear el recurso, en nuestro caso el fichero, que queremos mantener común para todos los procesos que se ejecuten en el mismo procesador.

Otro de los problemas que tenía la librería secuencial era que producía las anotaciones sobre una variable de tipo `IORef`. Al igual que las variables `MVar`, hemos comprobado que ese tipo de variables tampoco funciona adecuadamente en un entorno paralelo, ya que corresponden con punteros a memoria, por lo que hemos decidido crear las anotaciones sobre un fichero del sistema. La principal desventaja de realizarlas sobre un fichero consiste en que retrasa un poco más el cómputo. Pero debido a que la depuración sólo se utiliza en la etapa de desarrollo no lo consideramos problemático.

Utilizando las funciones sobre semáforos que ofrece el sistema, así como escribiendo las anotaciones sobre un fichero, hemos creado nuestra propia librería simple, implementada en C, que se encarga del manejo de los semáforos, cuyas funciones son las que se detallan a continuación:

`int createSemaphore(int idsem)`: Crea el semáforo identificado por `idsem` y lo inicializa a libre. Si este existe lo destruye y lo crea inicializado a libre.

`void lockSemaphore(int idsem)`: Bloquea el semáforo correspondiente con `idsem`. Si el semáforo no existe, este se crea. De esta manera, cuando un proceso llega a un procesador nuevo y quiere generar observaciones, no se tiene que preocupar por crear el semáforo.

`void unlockSemaphore(int idsem)`: Desbloquea el semáforo correspondiente con `idsem`.

`void deleteSemaphore(int idsem)`: Elimina el semáforo correspondiente con `idsem`.

Por otro lado, las anotaciones de observación que se producen en dicha librería necesitan un generador de identificadores para el *portId* (véase la Sección 3.4.3). La librería original utiliza un entero referenciado con una variable de tipo `IORef` para este propósito. Como ya se ha comentado anteriormente, este tipo de variables no funciona correctamente en paralelo. Por tanto, hemos creado una librería en C, basada en la librería de los semáforos vista anteriormente, que nos permite generar estos valores. Las funciones necesarias han sido las siguientes:

`int initUniq(int iduniq)`: Crea un generador de números con la identidad `iduniq` inicializado a 1.

`int getAndIncUniqSem(int idsem, int iduniq)`: Obtiene el número del generador `iduniq`, bloqueando el generador con el semáforo `idsem` e incrementando su valor en 1. De esta manera se consigue que varias hebras que traten de coger el número a la vez esperen.

`void deleteUniq(int iduniq)`: Elimina el generador de números identificado por `iduniq`.

Para realizar las tareas de manera uniforme, al igual que con la función `getAndIncUniqSem` que se encarga de obtener el identificador bloqueando y desbloqueando el semáforo, hemos creado el siguiente procedimiento encargado de escribir el *string* `str` en el fichero referenciado por `file`, bloqueando y desbloqueando adecuadamente el semáforo referenciado por `idsem`: `void writeStrOnFileSem(int idsem, char* file, char* str)`. De esa manera desde Haskell sólo será necesario mandar la anotación que se desea escribir en el fichero `file` a dicho procedimiento.

Aplicando estas leves modificaciones a la librería se consigue que funcione en paralelo y produzca las observaciones de las expresiones anotadas como observadas tanto en GpH como en Eden. Debemos tener en cuenta que ambos lenguajes son paralelos, por consiguiente, cuando las hebras de GpH o los procesos de Eden realizan una llamada al sistema, dicha llamada se ha realizado en el computador en que se encuentra dicha hebra. En consecuencia, cuando se están evaluando en paralelo varias hebras observables se crea en cada computador un fichero protegido por un semáforo del sistema operativo del computador en el que se crea dicha hebra. En otras palabras, se crea un fichero en cada computador donde se estén realizando las observaciones y se crea un semáforo que protege dicho fichero, distinto en cada computador.

7.4.2. Modificando la librería de observaciones para el análisis de especulación

Como se ha comentado anteriormente, al evaluar un programa en paralelo y utilizar la librería de observaciones se produce un fichero en cada procesador en que se están ejecutando los procesos. Para poder comparar las observaciones de los diferentes ficheros en relación al tiempo es necesario modificar la librería de observaciones para que de alguna manera se produzca una firma temporal en cada anotación de observación. De esta manera será sencillo comparar las observaciones producidas en los diferentes ficheros de observación y generar a partir de ellas un seguimiento temporal de los acontecimientos. Si bien es cierto que los relojes de cada procesador pueden no estar sincronizados y cuando se analicen las observaciones de los diferentes ficheros no se pueda obtener una secuencia exacta de acontecimientos, ya que estos puede que se encuentren

desplazados con respecto a los de otro procesador, no obstante, la aproximación que obtendremos será suficiente para el análisis de especulación.

Como se puede intuir, la modificación necesaria será escasa, ya que lo único necesario será añadir las marcas temporales de observación. La forma sencilla de realizar esto es hacerlo a través de una modificación en la librería de C. Recordaremos que todas las observaciones se producían a través del procedimiento `void writeStrOnFileSem(int idsem, char* file, char* str)` que se encargaba de escribir la observación `str` en el fichero `file` bloqueando y desbloqueando adecuadamente el semáforo `idsem` para proteger el fichero de observaciones adecuadamente. Lo único necesario será que ahora este mismo proceso en vez de escribir sólo la marca de observación deberá escribir la firma temporal que indica cuándo se produce dicha observación. Para realizar esta firma temporal nos hemos basado en la librería `sys/timeb.h` de C. Más concretamente hemos utilizado la función `int ftime(struct timeb *fecha)` que está basada en la siguiente estructura:

```
struct timeb {
  time_t time;
  unsigned short millitm;
  short timezone;
  short dstflag;
};
```

Para la firma temporal nos conformamos con tener los milisegundos en los que se produce una observación, ya que los segundos nos parecen escasos (un programa puede finalizar en menos de un segundo) y los nanosegundos nos parecen demasiado precisos (para realizar el análisis de especulación no es necesario obtener con tanta precisión el momento en que se han producido las observaciones, recordemos que dicha precisión sólo se mantiene dentro de cada fichero). El cálculo de los milisegundos se realiza de la siguiente forma `1000.0*fecha.time +fecha.millitm`. De esta forma este número indica el número de milisegundos al que corresponde la `fecha`.

Rendimiento y demora debido al análisis especulación

Como ya se ha comentado previamente el análisis de especulación introduce una demora en la evaluación del programa. Esta demora es debida a que las anotaciones se producen sobre un fichero común y a que los procesos pueden bloquearse sobre dicho fichero, perdiendo tiempo en la obtención de sus resultados. Tal y como ya se ha comentado, dicha demora no es preocupante, ya que este análisis se realiza cuando el implementador está desarrollando la aplicación correspondiente.

Sin embargo, si este análisis conllevara mucho tiempo podría alterar sustancialmente el comportamiento de los programas bajo observación. Por este motivo hemos decidido realizar un estudio del consumo de tiempo de las observaciones. Este análisis se ha realizado sobre el problema `linSolv`. El aumento del tiempo con respecto a la ejecución sin observaciones ha sido un 2.35 %. En total se han realizado 5470 anotaciones de observaciones sobre el fichero en un tiempo de $2,234$ segundos. Por consiguiente, cada anotación se ha producido en un tiempo que ronda los $0,4$ milisegundos, algo menos de media milésima. Con esto queda comprobado que las observaciones no realizan una demora sustancial sobre la ejecución del programa. De esta manera la utilización de esta herramienta parece muy adecuada para analizar los programas sin alterarlos susticialmente.

Capítulo 8

Incorporando facilidades de depuración en la semántica de GpH y Eden

Una vez que ya poseemos un depurador funcionando en los entornos paralelos GpH y Eden, creemos necesario dar una semántica a la depuración en dichos entornos. Esta semántica no va a estar guiada por la implementación, sino por las intuiciones sobre el comportamiento que debería tener. Por tanto, nuestra tarea en este capítulo será extender con facilidades de depuración la semántica de dos lenguajes paralelos, GpH y Eden.

Ambos lenguajes se basan en Haskell y poseen propiedades muy interesantes. GpH está basado en que el programador anote las expresiones que pueden calcularse en paralelo, mientras que Eden está basado en la creación de procesos por el programador.

Como dichos lenguajes están basados en Haskell, gran parte de la semántica definida en el Capítulo 6 va a ser aprovechada para definir las semánticas que desarrollaremos en este capítulo. Además, la implementación vista en el capítulo anterior nos servirá en ciertos momentos como intuición inicial para desarrollar la semántica. Como ya hemos comentado, con esto no estamos queriendo decir que copiaremos literalmente el comportamiento de dicha implementación. Sino que nos plantearemos varias opciones semánticas y adoptaremos la que consideremos más interesante, sin que la implementación nos obligue a elegir una concreta. También estudiaremos la semántica de dicha implementación y la compararemos con la que nosotros consideramos más adecuada.

No sólo incluiremos en la semántica de GpH y Eden las características de la depuración basada en observaciones, sino que previamente las adaptaremos a la semántica de Sestoft, incluyendo para ello las expresiones necesarias y definiendo el comportamiento de dichas expresiones.

En resumen, el trabajo desarrollado en este capítulo ha sido el siguiente:

1. Adaptar las semánticas de GpH y Eden, basadas en la semántica de Launchbury, a las ideas presentadas por Sestoft. De esta manera ha sido necesario cambiar el renombramiento de las variables de la regla *Var* a la regla *Letrec*, para conseguir una distinción clara entre los punteros (variables libres) y las variables del programa (variables ligadas).

2. Incluir los constructores, las expresiones **case** y los valores primitivos en ambas semánticas.

3. Adaptar la notación a un marco común.

4. Incluir la depuración en ambas semánticas.

Como es lógico, el último punto es el que consideramos más relevante e importante dentro del trabajo de esta tesis.

El trabajo expuesto en este capítulo ha dado lugar a la publicación [ELRH07].

8.1. GpH-Eden: Lenguaje core

Aunque GpH y Eden extienden el lenguaje funcional Haskell en diferente manera, ambos comparten un núcleo común para dichos lenguajes. La sintaxis del lenguaje correspondiente al núcleo común a ambos se presenta en la Figura 8.1 y será denotada a partir de ahora como lenguaje GpH-Eden Core. Dicho lenguaje es una extensión del que vimos en el Capítulo 6 (un lambda cálculo extendido con expresiones **let** recursivas, aplicaciones de constructores, expresiones **case** y valores primitivos) al que ahora le hemos añadido las expresiones de GpH (tanto la secuencial como la composición paralela) y las expresiones Eden (la instanciación de procesos). Las expresiones de GpH y Eden se transforman en expresiones GpH-Eden Core a través de una *fase de normalización*, que introduce las expresiones **letrec** necesarias, tal y como sucedía en el lenguaje de partida. Los *streams de Eden* son considerados como constructores: $(x_1 : x_2)$ es un constructor de aridad 2 ($Cons\ x_1\ x_2$) y [] es un constructor de aridad 0 (Nil), por lo tanto, pueden aparecer en expresiones **case**. Hemos utilizado la misma notación que las listas de Haskell, ya que en Eden realmente se corresponden con listas normales y son tratados como tales, salvo que la comunicación de dichas listas entre los procesos se realiza de elemento en elemento. Otra cuestión importante a resaltar es que en este lenguaje no aparecen las abstracciones de proceso de Eden. Esto se debe a que a nivel semántico no existe ninguna diferencia entre los procesos y las λ-abstracciones.

Al igual que en la extensión de la versión secuencial, el programador puede anotar cualquier variable como observable. En particular, se pueden marcar las variables que hacen referencia a la abstracción del proceso. Recalcaremos que al igual que sucedía en la versión secuencial, en este lenguaje paralelo aparecen dos *expresiones internas*. Dichas expresiones son : $p^{@(r,s)}$ y $\lambda^{@[\overline{(r_i,s_i)}]}x.e$. Fíjese, que la λ-abstracción va anotada con una lista de observadores, ya que puede estar observada desde varios puntos. Al programador no se le permite escribir este tipo de expresiones: dichas expresiones son expresiones auxiliares que aparecen como resultado de aplicar las reglas semánticas.

A continuación utilizaremos la notación siguiente: $x, y, p, q, l, t, ch, ch_o, ch_i \in Var$ para las "variables ordinarias", x, y son variables de programa, mientras que p, q, l, t son variables libres creadas de forma dinámica (punteros), y ch_i se corresponde con canales de entrada en Eden (del padre al hijo), ch_o se corresponde con los canales de salida en Eden (del hijo al padre) y ch se corresponderá a un canal de Eden de los descritos anteriormente, pero donde no nos interesa si es de entrada o salida. Utilizaremos la notación w para las formas normales débiles de cabeza. La notación $\overline{P_j}$ se utilizará para la repetición de un cierto *patrón* P indexado por la variable j. Por tanto, la notación **letrec** $\overline{x_i = be_i}$ **in** e significa: **letrec** $x_1 = be_1, \ldots, x_n = be_n$ **in** e.

8.2. Sistema de transiciones

Siguiendo las ideas de [BKT00, HO02], modelamos la evaluación del estado de un proceso como su conjunto de ligaduras de variables a expresiones (clausuras), es decir, un *heap*. Además,

		-- Expresiones	
e	$\rightarrow$	x	-- variable
	$\mid$	$\lambda\ x.e$	-- abstraccion lambda
	$\mid$	$x\ y$	-- aplicación
	$\mid$	**letrec** $\overline{x_i = be_i}$ **in** e	-- let recursivo
	$\mid$	$C\ \overline{x_i}$	-- aplicación de constructores
	$\mid$	**case** x **of** $\overline{C_i\ \overline{x_{ij}} \rightarrow e_i}$	-- expresión case
	$\mid$	$prim$	-- valores primitivos
	$\mid$	$op\ \overline{x_i}$	-- operador primitivo saturado
	$\mid$	x `` `seq` `` y	-- evaluación secuencial de GpH
	$\mid$	x `` `par` `` y	-- evaluación paralela de GpH
	$\mid$	x `#` y	-- creación de proceso de Eden
		-- Ligaduras	
be	$\rightarrow$	e	-- expresiones
	$\mid$	$x^{@str}$	-- **variables observadas**
	$\mid$	$p^{@(r,s)}$	-- **puntero observado (interna)**
	$\mid$	$\lambda^{@\overline{[(r_i,s_i)]}}x.e$	-- **abstracción lambda observada (interna)**
		-- Formas normales débiles de cabeza (*whnf*)	
w	$\rightarrow$	$C\ \overline{x_i}$	-- aplicación de constructores
	$\mid$	$prim$	-- valores primitivos
	$\mid$	$\lambda\ x.e$	-- abstracción lambda
	$\mid$	$\lambda^{@\overline{[(r_i,s_i)]}}x.e$	-- **abstracción lambda observada (interna)**
	$\mid$	L	-- *streams* de Eden
		-- Primitivos	
$prim$	$\rightarrow$	$int(1,\ 2,\ \ldots)$	-- enteros primitivos
	$\mid$	$\ldots$	-- otros
		-- Operaciones sobre primitivos	
op	$\rightarrow$	$+$	-- suma
	$\mid$	$\ldots$	-- otras
		-- Streams de Eden	
L	$::=$	$[]$	-- *stream* vacío
	$\mid$	$(x_1 : x_2)$	-- *stream* no vacío

Figura 8.1: Lenguaje core GpH-Eden-Observado

cada ligadura se considera una hebra en potencia y, por tanto, tiene asociada una etiqueta indicando su estado: $p \overset{\alpha}{\mapsto} e$, donde $\alpha ::= I|A|B|R$ se corresponde respectivamente con

I: *Inactiva*, o aún sin demandar o completamente evaluada,

A: *Activa*, demandada y en ejecución,

B: *Bloqueada*, demandada pero esperando al resultado del valor de otra ligadura,

R: *Ejecutable*, demandada y esperando a que haya un procesador disponible.

La distinción entre *Activa* y *Ejecutable* es exclusiva de GpH, ya que en la semántica de GpH se tiene en cuenta el número de procesadores disponibles en tiempo de ejecución.

Definición 8.1

1. Se define una *ligadura*, $p \overset{\alpha}{\mapsto} e$, donde α es un estado, p un puntero y e una expresión.
2. Se define un *heap*, H, como un conjunto finito de ligaduras.
3. Dado un *heap* $H = \{p_1 \overset{\alpha_1}{\mapsto} e_1, \ldots, p_n \overset{\alpha_n}{\mapsto} e_n\}$, el *dominio de* H, $dom(H)$, es el conjunto de las variables en él ligadas, $\{p_1, \ldots, p_n\}$.
4. Un *proceso* es un par $\langle id, H \rangle$, donde p es un identificador de proceso, y H es un *heap*.
5. Un *sistema*, S, es un conjunto no vacío de procesos. Un *sistema finito* es un sistema con una cantidad finita de ligaduras.

Nótese que todo sistema finito contiene un número finito de procesos.

□

Intuitivamente, un sistema representa un conjunto de procesos que se evalúan en paralelo. La evaluación de un programa en GpH-Eden Core será representada por una secuencia de sistemas, regida por las reglas de transición que introducimos a continuación. Dentro de cada *heap* las ligaduras se pueden ejecutar concurrentemente.

En las próximas secciones, con la idea de abordar la reescritura de multiples reglas de transición, permitiremos a una ligadura aparecer con varias etiquetas, correspondientes con las diferentes posibilidades admitidas por dicha regla. Por tanto, si por ejemplo $p \overset{IAB}{\mapsto} e$ aparece en la parte izquierda de la regla, y $p \overset{ABA}{\mapsto} e'$ en la parte derecha de la regla, significa que la hebra correspondiente a la clausura $p \mapsto e$ se convierte en activa en el caso de que se encontrara previamente inactiva o bloqueada, mientras que se convierte en bloqueada si previamente se encontraba activa. El conjunto $dom(H)$ contiene las variables de la parte izquierda del *heap* H. Además, la notación $H + \{p \overset{\alpha}{\mapsto} e\}$ significa que el *heap* H es extendido con la nueva ligadura $p \overset{\alpha}{\mapsto} e$, mientras que $H : p \overset{\alpha}{\mapsto} e$ significa que la ligadura p es la que guía la aplicación de la evaluación de la regla correspondiente; en ambos casos $p \notin dom(H)$.

A la hora de evaluar un programa e, el heap inicial se corresponde con el siguiente: $H_0 = \{p_{main} \overset{A}{\mapsto} e\}$. Se asume que p_{main} se corresponde con una variable fresca (puntero). Por tanto, e no contiene la variable p_{main}.

Definición 8.2 (Expresiones bloqueadas) Una ligadura como $p \overset{B}{\mapsto} e$ indica que la expresión e se encuentra bloqueada y esperando al resultado de otro cómputo. Consideraremos que la expresión e se encuentra bloqueada en la variable x y lo denotaremos de la siguiente forma: $e \in \mathsf{ble}(x)$, si la expresión e tiene alguno de los siguientes aspectos: $\{x,\ x\ y,\ \mathbf{case}\ x\ \mathbf{of}\ alts,\ x^{@(r,s)},\ op\ x\ y,\ op\ y\ x,\ x\ \texttt{`seq`}\ y\}$.

□

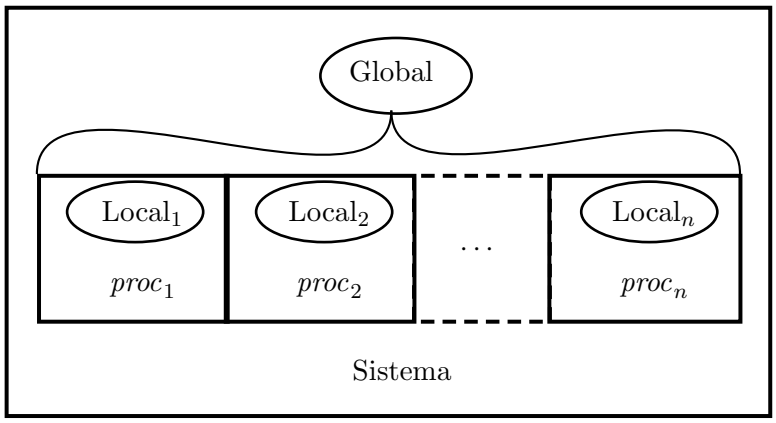


Figura 8.2: Dos niveles de transiciones

Como es usual en los lenguajes paralelos y concurrentes [BKT00, HO02], la semántica operacional de GpH y Eden implica dos niveles en el sistema de transiciones (véase la Figura 8.2):

1. Uno de bajo nivel para manejar el comportamiento *local* de los procesos. A este nivel se desarrollan los cómputos locales puramente funcionales en cada proceso, como la β-reducción, la reducción de las expresiones **letrec**, **case**, etc.

   El comportamiento local de GpH y Eden es básicamente el mismo, que es muy similar al comportamiento secuencial visto en el Capítulo 6. La única diferencia radica en que ahora consideraremos las ligaduras en el *heap* como hebras y, por tanto, llevarán una anotación indicando en qué estado se encuentran (activas, ejecutables, etc.), es decir, una configuración se corresponderá con un *heap* que tiene muchas hebras activas (posibles expresiones de control). Por otro lado, las diferencias entre Eden y GpH se corresponden a que las hebras de Eden no realizan ninguna distinción entre el estado ejecutable y el de activo.

2. Otro a nivel superior para describir el comportamiento *global* del sistema. A este nivel se desarrollan las interacciones entre los procesos del sistema, es decir, la creación de procesos, las comunicaciones entre ellos, etc.

El nivel de abstracción de la semántica de GpH y Eden no es tan elevado como el de las semánticas de Launchbury y Sestoft. Tanto la semántica de GpH como la de Eden consisten en una semántica operacional de paso corto. Se puede considerar como una semántica a medio nivel entre la semántica y las máquinas abstractas de Sestoft. La única diferencia con respecto a las máquinas de Sestoft es que no poseen pila y en este caso la expresión de control es un vínculo más dentro del *heap*, por ese motivo se presentará con una notación más cercana a la máquina abstracta. Comparada con la semántica de Sestoft, se puede observar que los pasos de evaluación de estas semánticas son más pequeños, pero sin embargo mantienen la idea de que una configuración evoluciona a otra.

8.2.1. Transiciones locales

(demanda)

si $e \notin whnf$ $$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\}$$

(valor)

$$H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q \longrightarrow H + \{q \overset{I}{\mapsto} w, p \overset{A}{\mapsto} w\}$$

(agujero negro)

$$H : p \overset{A}{\mapsto} p \longrightarrow H + \{p \overset{B}{\mapsto} p\}$$

(app-demanda)

si $e \notin whnf$ $$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ l \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\ l\}$$

(β-reducción)

$$H + \{q \overset{I}{\mapsto} \lambda x.e\} : p \overset{A}{\mapsto} q\ l \longrightarrow H + \{q \overset{I}{\mapsto} \lambda x.e, p \overset{A}{\mapsto} e[l/x]\}$$

(letrec)

$frescas(\overline{q_i})$ $$H : p \overset{A}{\mapsto} \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \longrightarrow H + \{\overline{q_i \overset{I}{\mapsto} e_i[\overline{q_i/x_i}]}, q \overset{A}{\mapsto} e[\overline{q_i/x_i}]\}$$

(case-demanda)

si $e \notin whnf$ $$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} \textbf{case } q \textbf{ of } alts \longrightarrow H + \{q \overset{RABR}{\mapsto} e,\ p \overset{B}{\mapsto} \textbf{case } q \textbf{ of } alts\}$$

(case-reducción)

$$H + \{q \overset{I}{\mapsto} C_k\ \overline{p_i}\} : p \overset{A}{\mapsto} \textbf{case } q \textbf{ of } \overline{C_i\ \overline{x_{ij}} \mapsto e_i} \longrightarrow H + \{q \overset{I}{\mapsto} C_k\ \overline{p_i},\ p \overset{A}{\mapsto} e_k[\overline{p_i/x_{kj}}]\}$$

(opP-demanda)

si $e_i \notin whnf$ y $i \in \{1 \ldots n\}$ $$H + \{q_i \overset{IABR}{\mapsto} e_i\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n \longrightarrow H + \{q_i \overset{RABR}{\mapsto} e_i,\ p \overset{B}{\mapsto} op\ \overline{q_i}^n\}$$

(opP-reducción)

$$H + \{\overline{q_i \overset{I}{\mapsto} m_i}^n\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n \longrightarrow H + \{\overline{q_i \overset{I}{\mapsto} m_i}^n,\ p \overset{A}{\mapsto} op\ \overline{m_i}^n\}$$

Figura 8.3: GpH-Eden Core: Reglas del transición local

Hemos dividido las transiciones en dos grupos: aquellas que manejan las expresiones ordinarias (Figura 8.3) y aquellas que manejan las observaciones (Figura 8.4). Las reglas de la Figura 8.3 expresan cómo se comporta la evaluación de la pereza bajo demanda. Si debemos evaluar una expresión en un entorno perezoso, su evaluación tiene que ser demandada. Esto se representa con una variable *activa*. Por dicho motivo, la *ligadura guía* en todos los casos se corresponde con una hebra activa. Las variables activas pueden estar ligadas a un valor (en ese caso su evaluación ha finalizado), una aplicación, una expresión **letrec**, o una expresión **case**.

Cuando la ligadura es activa y se encuentra ligada a otra variable, las posibilidades son las siguientes:

- Ambas variables se corresponden con la misma: entonces dicha ligadura debe ser bloqueada (regla **agujero negro**).
- Ambas variables son diferentes. En este caso pueden suceder dos situaciones:

- La última variable se encuentra ligada a un valor: entonces este debe ser copiado a la variable bajo evaluación (regla **valor**).
- La última se encuentra ligada a una expresión que aún no se encuentra evaluada: entonces la variable bajo evaluación debe ser bloqueada mientras que la variable bajo evaluación cambia el estado a *ejecutable* (en GpH) o *a*ctiva (en Eden) en caso de que ésta estuviera *inactiva* (**demanda**).

Si la variable se encuentra ligada a una aplicación se nos presentan dos casos: la variable correspondiente con su cuerpo se encuentra ligada con una λ-abstracción o con una expresión que no se encuentra en *whnf*. En el primer caso la β-reducción debe llevarse a cabo (regla **β-reducción**); en el segundo caso la evaluación del cuerpo debe ser demandada, i.e., y la ligadura se convierte en ejecutable (regla **app-demanda**).

De forma similar, cuando evaluamos una expresión **case**, si la variable se encuentra ligada a un constructor entonces la reducción debe ser realizada (regla **case-reducción**). En caso contrario, su evaluación debe ser demandada (regla **case-demanda**).

Finalmente, la evaluación de una expresión let introduce nuevas ligaduras que se corresponden con las variables locales (renombradas) (regla **letrec**).

Estas reglas son similares a las reglas secuenciales del Capítulo 6, con las siguientes salvedades:

- Ahora no poseemos una expresión en el control, sino que las ligaduras activas son las que pueden evolucionar, en definitiva las que llevan el control.
- Varias reglas semánticas se han dividido en dos reglas independientes, similares a las que aparecían en la máquina *Mark-1* de Sestoft, necesarias para definir el comportamiento de la regla semántica correspondiente. Por esta razón dicha semántica se considera una semántica de paso pequeño. (Ej. La regla *Case* se ha dividido en las reglas semánticas **case-demanda** y **case-reducción**, que son similares a las reglas *case1* y *case2* de la máquina *Mark-1* respectivamente.)

Reglas locales con observación

El siguiente conjunto de reglas, que aparece en la Figura 8.4, expresa cómo progresa la evaluación de las marcas de observación bajo la pereza tanto en GpH como en Eden. Nótese que también este conjunto de reglas es más cercano a las reglas de la Figura 6.6 que a las reglas de la semántica presentadas en la Figura 6.3. Tal y como hicimos tanto en la semántica como en la máquina secuencial, para trabajar con observaciones debemos añadir un fichero a las reglas locales. Dicho fichero almacena las observaciones que se realizan en la evaluación de un programa para posteriormente poder ser tratadas y mostradas en una forma más legible. Por tanto, las transiciones adquieren el siguiente aspecto: $H : p \overset{A}{\mapsto} e \looparrowright f \longrightarrow H' \looparrowright f'$, que significa que "la evaluación de la hebra activa $p \overset{A}{\mapsto} e$ transforma el *heap* $H + \{p \overset{A}{\mapsto} e\}$ en H' y añade las observaciones al fichero f generando por tanto (f')". La información es añadida al fichero de forma secuencial. Mantenemos la notación $f \circ \langle ann \rangle$ para indicar que añadimos la anotación *ann* en una nueva línea al final del fichero f. Deberíamos haber introducido el fichero en el primer conjunto de reglas (Figura 8.3) para mostrarlas completas. Sin embargo, como ninguna de ellas modifica el fichero, hemos preferido omitirlo para hacerlas mas legibles.

Al igual que en la semántica, nuestras anotaciones no tendrán exactamente la misma forma que las que realiza Hood (véase la Sección 3.4.3). Nuestras anotaciones mantienen el mismo

(observ)

$$H : p \overset{A}{\mapsto} q^{@str} \rightarrowtail f \;\longrightarrow\; H + \{p \overset{A}{\mapsto} q^{@(length\ f,0)}\} \rightarrowtail f \circ \langle 0\,0\ Observe\ str\rangle$$

(valor@L)

$$H + \{q \overset{I}{\mapsto} \lambda x.e\} : p \overset{A}{\mapsto} q^{@(r,s)} \rightarrowtail f \;\longrightarrow\; H + \{q \overset{I}{\mapsto} \lambda x.e,\ p \overset{A}{\mapsto} \lambda^{@[(r,s)]} x.e\} \rightarrowtail f$$

(valor@LO)

$$H + \{q \overset{I}{\mapsto} \lambda^{@obs} x.e\} : p \overset{A}{\mapsto} q^{@(r,s)} \rightarrowtail f \;\longrightarrow\; H + \{q \overset{I}{\mapsto} \lambda^{@obs} x.e,\ p \overset{A}{\mapsto} \lambda^{@(r,s):obs} x.e\} \rightarrowtail f$$

(valor@C)

$$\begin{array}{l} frescas(\overline{q_i})\ H + \{q \overset{I}{\mapsto} C\ \overline{p_i}^k\} : p \overset{A}{\mapsto} q^{@(r,s)} \rightarrowtail f \;\longrightarrow \\ \qquad \longrightarrow\; H + \{q \overset{I}{\mapsto} C\ \overline{p_i}^k,\ \overline{q_i \overset{I}{\mapsto} p_i^{@(length\ f,i)}},\ p \overset{A}{\mapsto} C\ \overline{q_i}^k\} \rightarrowtail f \circ \langle r\ s\ Cons\ k\ C\rangle \end{array}$$

(agujero negro@)

$$H : p \overset{A}{\mapsto} p^{@(r,s)} \;\longrightarrow\; H + \{p \overset{B}{\mapsto} p^{@(r,s)}\}$$

(demanda@)

$$\text{si } e \notin whnf\ H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q^{@(r,s)} \rightarrowtail f \;\longrightarrow\; H + \{q \overset{RABR}{\mapsto} e,\ p \overset{B}{\mapsto} q^{@(r,s)}\} \rightarrowtail f \circ \langle r\ s\ Enter\rangle$$

(β-reducción@)

$$frescas(t, l')\ H + \{q \overset{I}{\mapsto} \lambda^{@[\overline{(r_i,s_i)}]} x.e\} : p \overset{A}{\mapsto} q\ l \rightarrowtail f \;\longrightarrow\; H + \left\{ \begin{array}{l} q \overset{I}{\mapsto} \lambda^{@[\overline{(r_i,s_i)}]} x.e, \\ t \overset{I}{\mapsto} e[l'/x], \\ l' \overset{I}{\mapsto} l^{@(length\ f,0)}, \\ p \overset{A}{\mapsto} t^{@(length\ f,1)} \end{array} \right\} \rightarrowtail f \circ \langle \overline{[(r_i\ s_i)]}\ Fun\rangle$$

Figura 8.4: GpH-Eden Core: Transiciones locales con observaciones

aspecto que la extensión semántica vista en el Capítulo 6, es decir, para la observación de las funciones $\langle \overline{[(observeParent_i\ observePort_i)]}\ Fun\rangle$ y para el resto de observaciones mantenemos la anotación $\langle observeParent\ \ observePort\ \ change\rangle$. Al igual que en dicha extensión, el *portId* se corresponderá con la línea del fichero donde se produce la anotación. Por tanto será un entero y tanto el *observeParent* como el *observePort* serán enteros que se referirán al número de línea donde la anotación fue almacenada. La función *length f* devolverá el número de líneas del fichero f y consideraremos que la primera línea en el fichero se corresponde con la línea 0.

A continuación describiremos en detalle cada regla.

Regla observ. Cuando tenemos que evaluar una clausura que está anotada con el *string* de observación *str*, tenemos que generar una anotación en el fichero de observaciones que posee el siguiente aspecto $\langle 0\,0\ Observe\ str\rangle$ y continuar evaluando la clausura, pero ahora con una anotación que hace referencia a su padre, en este caso $(length\ f, 0)$.

Regla valor@L y valor@LO. Este caso corresponde con una ligadura activa con $p \overset{A}{\mapsto} q^{@(r,s)}$,

donde q está siendo observada y se encuentra ligada a una función. Si esta función ya posee alguna marca de observación (regla **valor@LO**) se añade la nueva marca de observación a dicha λ-abstracción ($\lambda^{@(r,s):obs}x.e$), sin embargo si dicha función no posee ninguna marca de observación (regla **valor@L**) creamos la siguiente λ-abstracción $\lambda^{@[(r,s)]}x.e$. Este tipo de λ-abstracción indica que la lambda se encuentra bajo las marcas de observación añadidas a la etiqueta @. A diferencia de la implementación que genera una marca de tipo *Enter*, nosotros en este caso no generamos una anotación de tipo *Enter* ya que ésta ha podido ser realizada debido a la aplicación de una regla **demanda@** previa.

Regla valor@C. Cuando $q^{@(r,s)}$ evalúa a un constructor, se crea la anotación $\langle r\ s\ Cons\ k\ C\rangle$ en el fichero de observaciones. Esto indica que la clausura cuyo padre es (r, s) ha sido reducida al constructor C (cuya aridad es k). Se generan nuevas clausuras apuntando a cada argumento de dicho constructor. Estas clausuras son anotadas indicando que están bajo observación. Además, en esta anotación debemos indicar su posición en el constructor y la referencia al padre (la línea en la que se ha realizado dicha anotación).

Regla agujero negro@ bloquea una clausura anotada bajo observación que necesita su valor para reducirse.

Regla demanda@. Si tenemos que evaluar una clausura del tipo $p \overset{A}{\mapsto} q^{@(r,s)}$, creamos una nueva anotación $\langle r\ s\ Enter\rangle$ indicando que hemos comenzado la evaluación de dicha clausura y seguimos evaluando.

Regla β-reducción@ Ésta es la regla fundamental para observar las funciones en nuestra semántica. Estamos evaluando la aplicación de una función observada. Primero generamos la anotación en el fichero indicando que estamos aplicando una función observada. Entonces marcamos su argumento como observable, y utilizamos $(length\ f, 0)$ como su padre. Para observar el resultado creamos una nueva clausura observada cuyo padre se corresponde con $(length\ f, 1)$. Los puertos son diferentes para recordar que uno es el argumento y el otro se corresponde con el resultado de la aplicación de la lambda.

Por último recalcaremos que, al igual que en la versión secuencial, al haber restringido los puntos donde la variable observada aparece, y debido a que nunca sustituimos una variable por un puntero observable, no es necesario especificar la aplicación a un puntero observado $p\ q^{@(r,s)}$.

8.2.2. Ejemplo de evaluación de transiciones locales

Antes de entrar en los detalles específicos de cada semántica creemos conveniente presentar un ejemplo del comportamiento local de las observaciones. Para ello consideraremos la misma expresión que en el Ejemplo 6.1 y veremos que se obtienen las mismas observaciones.

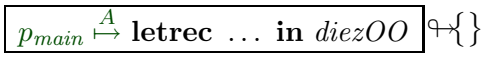
Ejemplo 8.1 Recordemos que este ejemplo es simple y que carece de interés, salvo por el hecho de que posee dos anotaciones de observación sobre la misma clausura. En él pretendemos llevar a cabo la observación de un único número desde dos marcas diferentes de observación. Partimos de la expresión inicial de Haskell, que se corresponde con una expresión válida en GpH y Eden:

```
observe "obs2" (observe "obs1" (10::Int))::Int
```

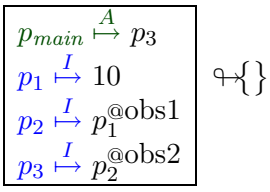
que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
  diez   = 10
  diez0  = diez@{obs1}
  diez00 = diez0@{obs2}
in diez00
```

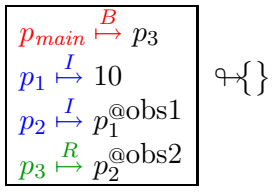
A continuación pasaremos a mostrar la reducción semántica completa para este ejemplo. No se mostrarán las reglas globales necesarias para la evolución del cómputo, tales como la regla de bloqueo de hebras que han alcanzado la forma normal y la regla de activación de las hebras que dejan de estar bloqueadas. En este caso, debido a que las configuraciones contienen un único *heap* y un fichero de observaciones, presentaremos la evolución mostrando completamente ambos elementos y resaltando en cada caso en verde las hebras activas.



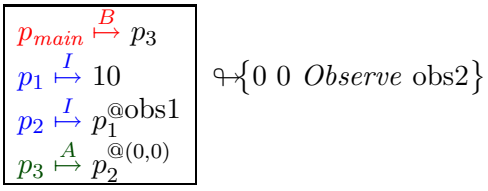
letrec



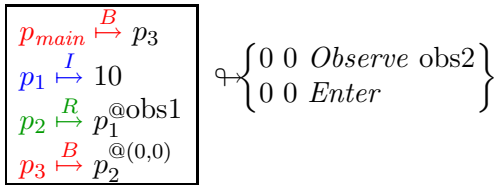
demanda

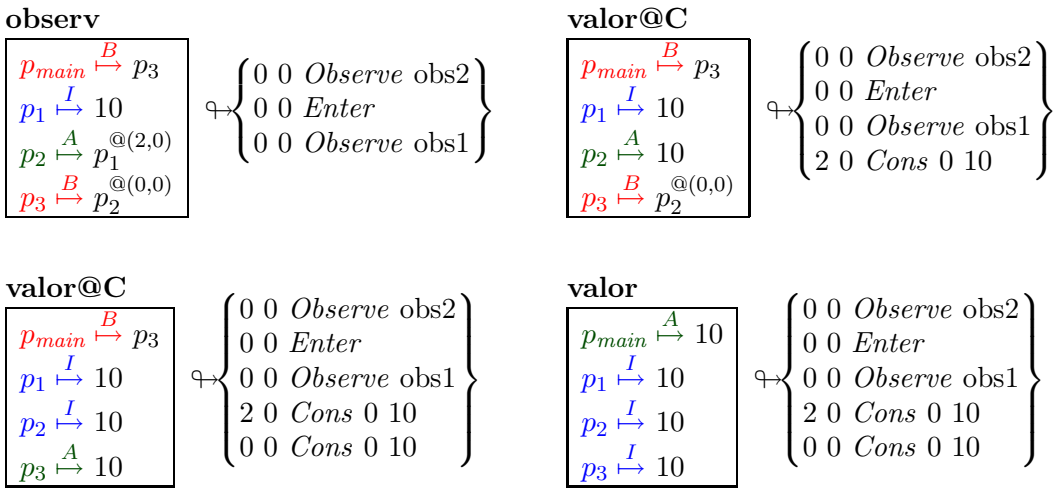


observ



demanda@





Por tanto, añadiendo los números de las líneas al fichero de observaciones obtenemos el siguiente fichero:

Línea	Observación
0	0 0 *Observe* obs2
1	0 0 *Enter*
2	0 0 *Observe* obs1
3	2 0 *Cons* 0 10
4	0 0 *Cons* 0 10

Analizando dicho fichero y comparándolo con el que se obtuvo en el Ejemplo 6.1 nos damos cuenta de que no posee una anotación de tipo *Enter* para la referencia $(2, 0)$ ya que dicha clausura se encontraba directamente en forma normal. Esto por un lado no es importante para la observación final y por otro lado se podría realizar un postprocesamiento del fichero para generar este tipo de anotaciones *Enter* para todas las clausuras que se hayan observado y no hayan generado dicha marca de observación. Por lo demás, el fichero es exactamente igual que el que apareció en dicho ejemplo y genera la misma observación, es decir:

```
-- obs1
        10
-- obs2
        10
```

□

8.3. Semántica formal de GpH

La evaluación de una expresión GpH Core precisa, en general, de la creación de varias hebras. Debido a que GpH no posee ni la noción de proceso, ni la noción de comunicación, el estado del programa consiste solamente en un único *heap*.

8.3.1. Transiciones locales de GpH

Antes de presentar las reglas de transición global para GpH, debemos explicar la semántica de los operadores: `‘seq‘` y `‘par‘`. Las reglas correspondientes se muestran en la Figura 8.5.

(seq)

si $e' \notin whnf$

$$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ \text{`seq`}\ q' \rightarrowtail\!\!\!\!\!\!\rightarrow f \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\ \text{`seq`}\ q'\} \rightarrowtail\!\!\!\!\!\!\rightarrow f$$

(rm-seq)

$$H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q\ \text{`seq`}\ q' \rightarrowtail\!\!\!\!\!\!\rightarrow f \longrightarrow H + \{q \overset{I}{\mapsto} w, p \overset{A}{\mapsto} q'\} \rightarrowtail\!\!\!\!\!\!\rightarrow f$$

(par)

$$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ \text{`par`}\ q' \rightarrowtail\!\!\!\!\!\!\rightarrow f \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{A}{\mapsto} q'\} \rightarrowtail\!\!\!\!\!\!\rightarrow f$$

Figura 8.5: GpH Core: Reglas de transición local

Las transiciones locales de GpH quedarán formadas por las reglas de las Figuras 8.3, 8.4 y 8.5. La secuencialidad de `‘seq‘` fuerza a demandar la evaluación de la variable de la parte izquierda cuando no se encuentra evaluada (no está en *whnf*) (regla **seq**). Cuando dicha variable es reducida a una *whnf*, la evaluación continúa con el cómputo de la parte derecha (regla **rm-seq**). Con respecto a la expresión `‘par‘`, ésta produce un cómputo potencialmente paralelo en la evaluación. Cuando un puntero p se encuentra ligado a una expresión, la evaluación del puntero p continúa con la variable de la derecha y crea una demanda sobre la variable de la izquierda en caso de que ésta no estuviera previamente demandada, de esta manera su estado pasa a ser ejecutable (regla **par**).

Fíjese que, gracias a la normalización, la generación de ligaduras se realiza solamente al evaluar expresiones **letrec** y en las reglas de observación **valor@C** y **β-reducción@**, pues tanto la aplicación, como la expresión **case**, como las expresiones `‘seq‘` o `‘par‘` y como la aplicación de los operadores primitivos hacen referencia a variables que han de poseer una ligadura en el *heap*. Sin embargo, es únicamente el operador `‘par‘` el que introduce paralelismo, pues la declaración local de variables solamente incluirá en el *heap* ligaduras inactivas, en tanto que el operador de paralelo aumenta las clausuras ejecutables sin necesidad de que exista demanda sobre ellas. No obstante, en las reglas de paso simple una ligadura en estado ejecutable no puede pasar a ser activa. Para poder realizar esta conversión se emplean las reglas de paso múltiple que veremos a continuación. Además, dicha conversión se realizará de acuerdo a la disponibilidad de recursos por parte del sistema, quedando así patente el hecho de que `‘par‘` es sólo una indicación de posible paralelismo, pero no una orden para realizarlo inexorablemente.

8.3.2. Transiciones globales de GpH

Es a este nivel donde veremos el comportamiento en paralelo de GpH. Como GpH posee un único *heap* común para todas las hebras, es bastante sencillo introducir las observaciones. Consideramos que a nivel semántico debería poseer un único fichero siguiendo la misma filosofía que para los *heaps* (recordaremos que en la implementación se produce un fichero por cada procesador, pero es fácil combinarlos para generar el fichero que se creará en la semántica). De esta manera, el fichero que almacena las observaciones se mantiene común para todas las hebras. Antes de mostrar las reglas es necesario definir la siguiente notación, que también será utilizada en Eden: las reglas globales vendrán definidas por la reiteración maximal de determinadas reglas de paso más pequeño.

$$\textbf{(paralela)}$$

$$\frac{H^A = \{p_i \overset{A}{\mapsto} e_i\}^n \quad \{H = H_U^i + H_M^i : p_i \overset{A}{\mapsto} e_i \multimap f_{i-1} \longrightarrow H_U^i + K^i \multimap f_i\}^n}{H \multimap f_0 \overset{par}{\Longrightarrow} \cap_{i=1}^n H_U^i \bigcup \cup_{i=1}^n K^i \multimap f_n}$$

$$\textbf{(desactivación)}$$

$$\frac{H_p^B = \{q_i \overset{B}{\mapsto} e_i\}^n}{H + \{p \overset{AR}{\mapsto} w, \overline{q_i \overset{B}{\mapsto} e_i}^n\} \multimap f \overset{desact}{\longrightarrow} H + \{p \overset{I}{\mapsto} w, \overline{q_i \overset{R}{\mapsto} e_i}^n\} \multimap f}$$

$$\textbf{(activación)}$$

$$\frac{|H^A| < \textbf{numeró de procesos} \wedge \ \mathsf{pre}(p_{main}, H + \{p \overset{A}{\mapsto} e\}) \in dom((H + \{p \overset{A}{\mapsto} e\})^A)}{H + \{p \overset{R}{\mapsto} e\} \multimap f \overset{act}{\longrightarrow} H + \{p \overset{A}{\mapsto} e\} \multimap f}$$

Figura 8.6: GpH Core: Planificación-paralelismo

Notación 8.1 *Sea S un sistema, y $\diamond$ un símbolo que representará el nombre de una regla. Para cada regla de paso simple $S \overset{\diamond}{\longrightarrow} S_1$ definimos una* regla de paso reiterado $S \overset{\diamond}{\Longrightarrow} S'$ *que viene dada por:*

1. $S \overset{\diamond}{\longrightarrow}^* S'$ *y,*
2. *no existe S'' tal que $S' \overset{\diamond}{\Longrightarrow} S''$.* □

Notación 8.2 *También debemos definir la notación $\overset{\diamond}{\Longrightarrow} \circ \overset{\square}{\Longrightarrow}$ que expresa la composición de las reglas $\diamond$ y $\square$; primeramente se aplica la regla $\square$ y acto seguido se aplica la regla $\diamond$.*

□

En [Hid04] se demostró que la aplicación de una regla de paso simple $\diamond$, a una ligadura concreta de un proceso específico, podía hacer posible la aplicación de esa misma regla $\diamond$ a otra ligadura (en el mismo o en otro proceso), pero nunca podría inhabilitar aplicaciones de $\diamond$ que eran posibles antes de la primera aplicación. Asimismo, también se demostró que la regla maximal correspondiente a cada regla de paso reiterado está bien definida.

La evolución local del sistema queda unívocamente determinada por el conjunto de hebras (activas) paralelas en H que pueden progresar. Utilizaremos la notación H^A para representar este conjunto de hebras activas del *heap* H, y $|H^A|$ como el cardinal de dicho conjunto. De forma análoga, la notación H_p^B representará el conjunto de hebras bloqueadas en el puntero p.

En la Figura 8.6 se presentan las tres reglas que manejan el comportamiento global del *heap*:

Regla paralela Esta regla hace que todas las hebras activas evolucionen en paralelo. La evolución se corresponde con una evolución local de cada regla y una mezcla posterior de los *heaps* resultantes.

Entraremos a explicarla en detalle. Estamos interesados en el comportamiento de todas las hebras activas del *heap* $p_i \overset{A}{\mapsto} e_i$. Para cada hebra, dividimos el *heap* en dos partes

$H = H^i_U + H^i_M$; H^i_U se corresponde con la parte del *heap* H que permanece sin cambios después de la aplicación de la regla local correspondiente, mientras que H^i_M se corresponde con la parte del *heap* que es modificada por la evolución de la hebra. Dicho *heap*, debido a la evolución, se transforma en K^i. Ahora, es necesario componer el *heap* final; este consiste en la parte del *heap* inicial que ha permanecido inalterable tras la evolución de todas las hebras $\cap_{i=1}^{n} H^i_U$, unido con el *heap* resultado de la ejecución de las hebras $\cup_{i=1}^{n} K^i$. [1]

Las anotaciones se escriben de forma secuencial en fichero. El fichero inicial se corresponde con el fichero f_0. En este fichero se escriben las anotaciones que produce la evaluación de la primera hebra $p_1 \overset{A}{\mapsto} e_1$, posteriormente sobre el fichero modificado se evalúa la segunda hebra y así hasta el final. Aparentemente las hebras se encuentran ordenadas, pero este orden no está previamente establecido, es más, la primera hebra que necesite el fichero será la que haga la primera modificación.

Nótese también que el fichero resultante es diferente si el orden de la evaluación de las hebras cambia. Lo realmente interesante es que todos los ficheros resultantes son coherentes con respecto a las observaciones y, por tanto, respecto al *heap* resultante. De tal manera que dicha regla se puede implementar evaluando las hebras de forma concurrente, únicamente sería necesario proteger el fichero por un semáforo que asegure la escritura en el fichero de forma secuencial, tal y como se realizó en la implementación.

Regla desactivación Cuando un puntero se encuentra ligado a un valor (*whnf*) en el *heap*, todas las ligaduras bloqueadas en él son desbloqueadas y su estado cambia a ejecutable y dicho puntero es desactivado. Esta regla no modifica el fichero de observaciones.

Regla activación El número de hebras activas en el sistema en GpH se encuentra limitado por el número de procesadores, que es una variable global del sistema. Cuando se activa una hebra (cambio del estado ejecutable al estado activo), se crea una preferencia sobre las hebras que son necesarias para la evaluación de la clausura *main*. Gracias al establecimiento de dicha precedencia se asegura que el programa termina, en caso de que este fuera a terminar. Dicha precedencia se encuentra definida por el conjunto de punteros $\mathsf{pre}(p_{main}, H)$, que se define así:

$$\mathsf{pre}(p, H) = \begin{cases} p, & \text{si } p \overset{AR}{\mapsto} e \in H \\ \mathsf{pre}(q, H), & \text{si } p \overset{B}{\mapsto} e \in H \text{ y } e \in \mathsf{ble}(q) \end{cases}$$

Esta regla tampoco modifica el fichero de anotaciones.

Si se combinan activación y desactivación maximales se obtiene una política de planificación de procesadores con la que se obtiene el máximo paralelismo entre las ligaduras del *heap* con respecto al número de procesadores que posee el sistema. En la regla semántica de planificación primero se lleva a cabo una liberación de procesadores, convirtiendo algunas ligaduras en inactivas, para después transformar en activas todas las que permita el número de procesadores disponibles. La planificación puede ser expresada utilizando la secuenciación:

$$\overset{sched}{\Longrightarrow} = \overset{act}{\Longrightarrow} \circ \overset{desact}{\Longrightarrow}$$

donde se aprecia que desactivación y activación se realizan en el orden preciso.

[1] La demostración de la corrección de esta regla puede encontrarse en [Hid04].

Terminamos especificando cómo se da un paso completo de cómputo en un *heap*: primero evolucionan todas las ligaduras activas y, posteriormente, se realiza una planificación de procesadores para optimizar los recursos.

El orden de aplicación de las reglas **parallel** y **schedule** es importante, pues si se realizaran en orden inverso podría darse el caso de tener un procesador ocupado en una ligadura activa cuya expresión sea ya una *whnf*. Por tanto la evolución global se define como sigue: primero se hace evolucionar el sistema en paralelo y posteriormente se ejecuta la planificación:

$$\Longrightarrow \;=\; \overset{sched}{\Longrightarrow} \;\circ\; \overset{par}{\Longrightarrow}$$

8.3.3. Ejemplo semántico de evaluación en GpH

Para comprender mejor el comportamiento de las observaciones en GpH creemos conveniente desarrollar la semántica de un ejemplo en GpH. En dicho ejemplo nos centraremos en los cómputos correspondientes con los pasos de evaluación de las observaciones. Para ello mostraremos la interacción del cómputo paralelo con las marcas de observación.

Ejemplo 8.2 En este ejemplo pretendemos observar la función Fibonacci evaluada en paralelo y observada con una marca de observación. El paralelismo lo conseguiremos debido a que evaluaremos los argumentos de forma paralela. Nótese que las observaciones también se producirán en paralelo, ya que la llamada recursiva de la función se realiza con la función *parfibO* que produce observaciones. En este ejemplo nos fijaremos únicamente en las observaciones y en la evolución paralela.

Partimos, por tanto, de la siguiente expresión en GpH:

```
main = parfib0 2

parfib0 = observe "parfib" parfib

parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `seq` (nf1+nf2))
           where nf1 = parfib0 (n-1)
                 nf2 = parfib0 (n-2)
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
  uno   = 1
  dos   = 2

  parfib  = \n. case n of
                  0 -> uno
                  1 -> uno
                  _ -> letrec
                                n1 = - n uno
                                n2 = - n dos
                                nf1 = parfib0 n1
                                nf2 = parfib0 n2
```

```
                                sum = + nf1 nf2
                                sol = nf1 `seq` sum
                            in   nf2 `par` sol

  parfib0 = parfib@parfib
in parfib0 dos
```

Nótese que el valor de n se corresponde con un entero. Recordemos que los enteros se consideran constructores de aridad 0. Pero con el fin de no colapsar las alternativas de la expresión **case** y debido a que la expresión **case** no posee alternativa por defecto, la simularemos con _ que significa que no nos interesa el valor de n.

Con el cálculo del Fibonacci de 2 será suficiente para ver la evolución en paralelo. Consideraremos que poseemos al menos dos procesadores. Pasemos a ver las reducciones más interesantes en este cómputo. Se parte inicialmente de la configuración siguiente:

$p_{main} \overset{A}{\mapsto}$ **letrec** ... **in** nf2 `par` sol $\rightarrowtail\{\}$

Letrec

$p_{main} \overset{A}{\mapsto} p_4\ p_2$
$p_1 \overset{I}{\mapsto} 1$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{I}{\mapsto} p_3^{@parfib}$

$\rightarrowtail\{\}$

observe

$p_{main} \overset{B}{\mapsto} p_4\ p_2$
$p_1 \overset{I}{\mapsto} 1$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{A}{\mapsto} p_3^{@(0,0)}$

$\rightarrowtail\{0\ 0\ \mathit{Observe}\ \text{parfib}\}$

valor@L

$p_{main} \overset{B}{\mapsto} p_4\ p_2$
$p_1 \overset{I}{\mapsto} 2$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{A}{\mapsto} \lambda^{@[(0,0)]} n.$ ...

$\rightarrowtail\{0\ 0\ \mathit{Observe}\ \text{parfib}\}$

β-reducción@

$p_{main} \overset{A}{\mapsto} p_6^{@(1,1)}$
$p_1 \overset{I}{\mapsto} 1$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{I}{\mapsto} \lambda^{@[(0,0)]} n.$ ...
$p_5 \overset{I}{\mapsto} p_2^{@(1,0)}$
$p_6 \overset{I}{\mapsto}$ **case** p_5 ...

$\rightarrowtail\left\{\begin{array}{l} 0\ 0\ \mathit{Observe}\ \text{parfib} \\ [(0\ 0)]\ \mathit{Fun} \end{array}\right\}$

demanda@

$p_{main} \overset{B}{\mapsto} p_6^{@(1,1)}$
$p_1 \overset{I}{\mapsto} 1$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{I}{\mapsto} \lambda^{@[(0,0)]} n.$ ...
$p_5 \overset{I}{\mapsto} p_2^{@(1,0)}$
$p_6 \overset{A}{\mapsto}$ **case** p_5 ...

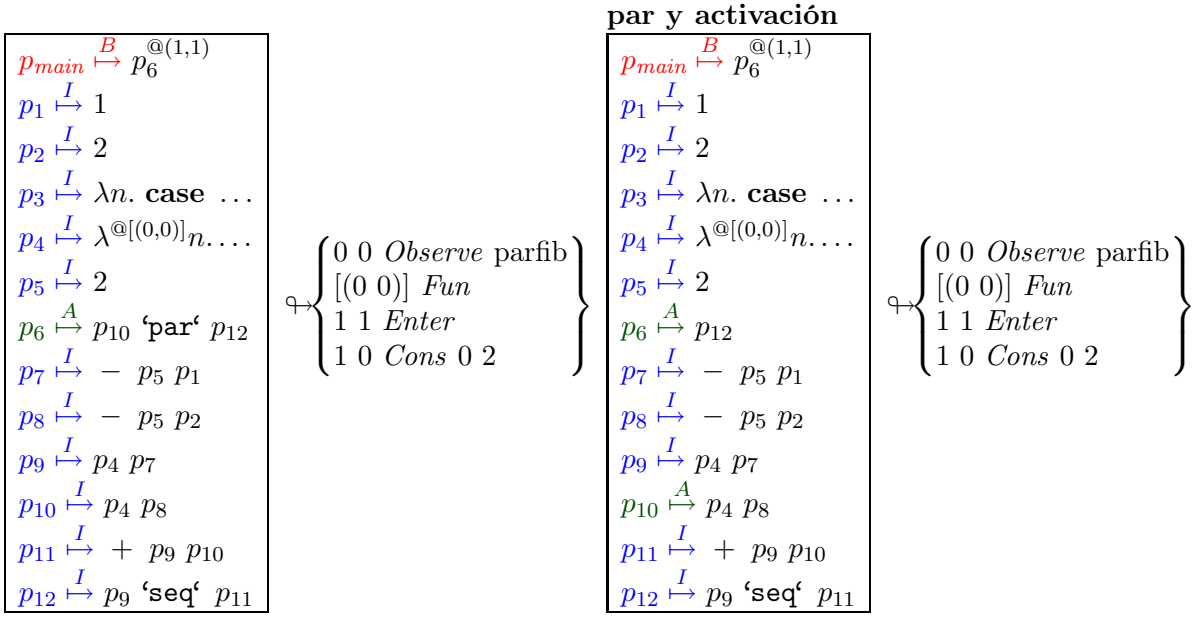
$\rightarrowtail\left\{\begin{array}{l} 0\ 0\ \mathit{Observe}\ \text{parfib} \\ [(0\ 0)]\ \mathit{Fun} \\ 1\ 1\ \mathit{Enter} \end{array}\right\}$

valor@C

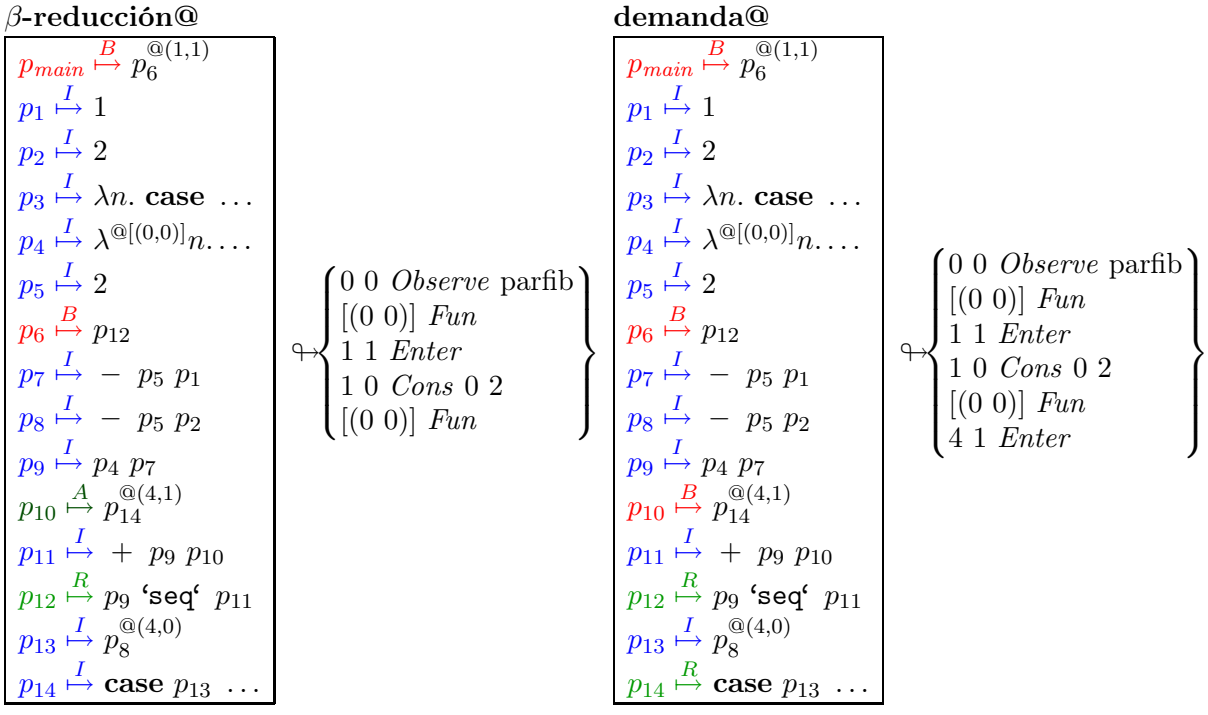
$p_{main} \overset{B}{\mapsto} p_6^{@(1,1)}$
$p_1 \overset{I}{\mapsto} 1$
$p_2 \overset{I}{\mapsto} 2$
$p_3 \overset{I}{\mapsto} \lambda n.$ **case** ...
$p_4 \overset{I}{\mapsto} \lambda^{@[(0,0)]} n.$ ...
$p_5 \overset{A}{\mapsto} 2$
$p_6 \overset{B}{\mapsto}$ **case** p_5 ...

$\rightarrowtail\left\{\begin{array}{l} 0\ 0\ \mathit{Observe}\ \text{parfib} \\ [(0\ 0)]\ \mathit{Fun} \\ 1\ 1\ \mathit{Enter} \\ 1\ 0\ \mathit{Cons}\ 0\ 2 \end{array}\right\}$

Veamos ahora el punto donde se generan dos hebras que pueden evolucionar en paralelo, es decir, la reducción de la expresión `‘par‘` (p_5). Veremos el *heap* inicial y la evolución:

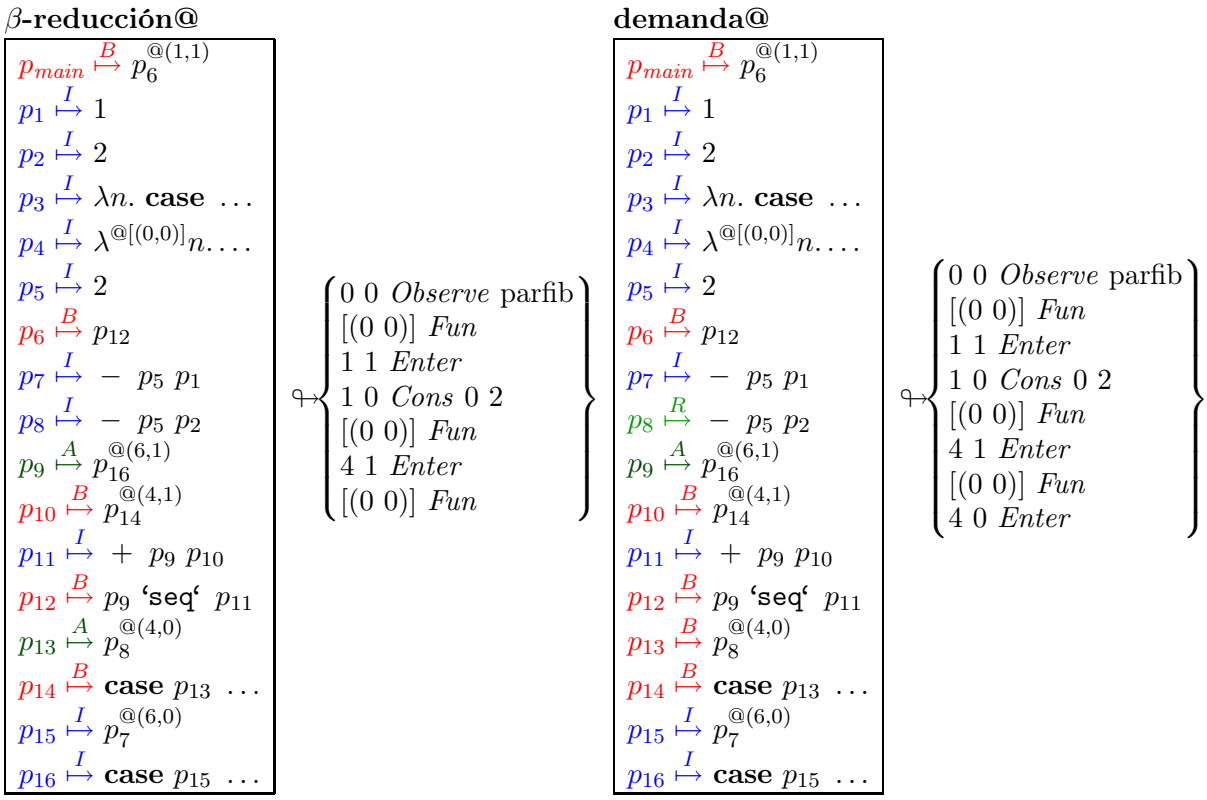


A partir de este momento se evaluarán en paralelo ambas hebras, ya que disponemos de dos procesadores. La evaluación de la hebra p_{10} producirá la siguiente observación sobre el fichero:



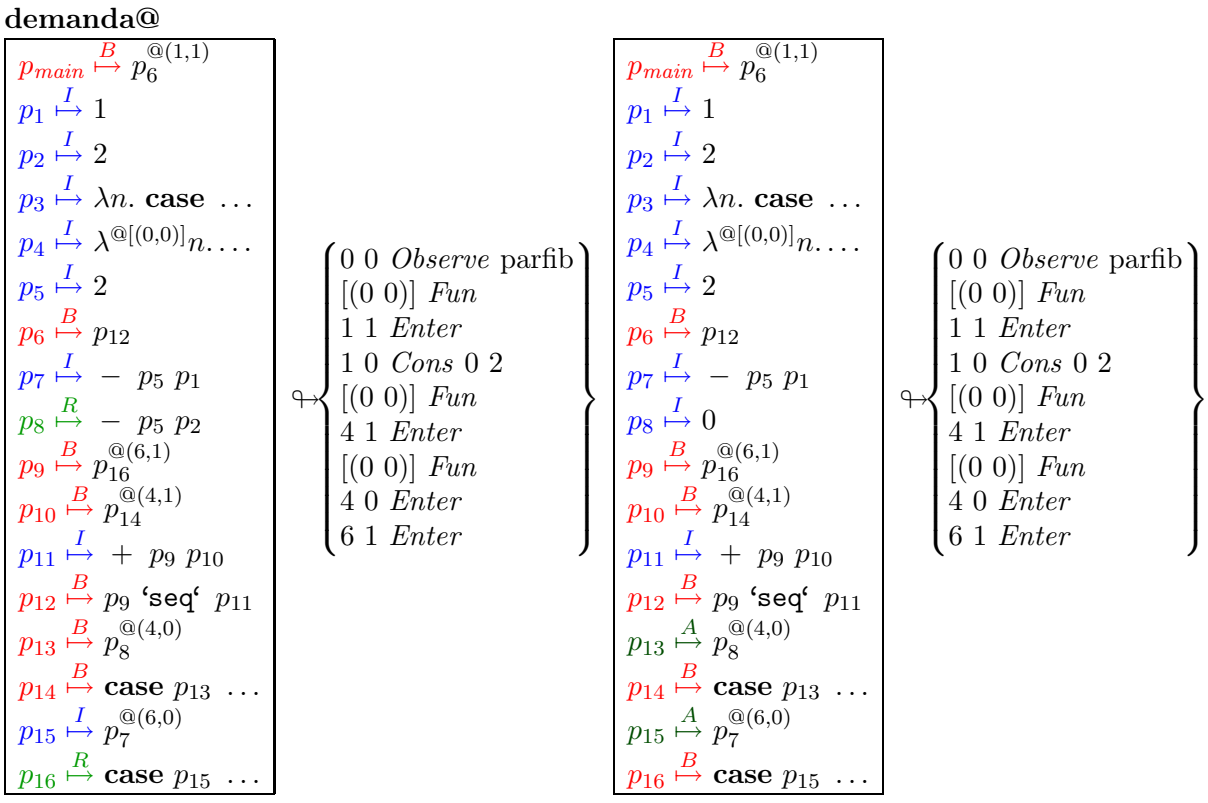
Es en este punto cuando se vuelve a aplicar la regla **paralela** que activa las hebras p_{12} y p_{14}. Esto provocará una demanda sobre las clausuras p_9 y p_{13}, que pasarán a estar activas

en paralelo tras una nueva aplicación de la regla **paralela**. En este punto se producirán dos observaciones: la correspondiente a la **β-reducción@** de la clausura p_9 y la correspondiente a la **demanda@** sobre la clausura p_{13}. Estas observaciones se pueden realizar en cualquier orden. Supondremos que primero se realiza la **β-reducción@** y posteriormente la **demanda@**. Veamos dichas reducciones:



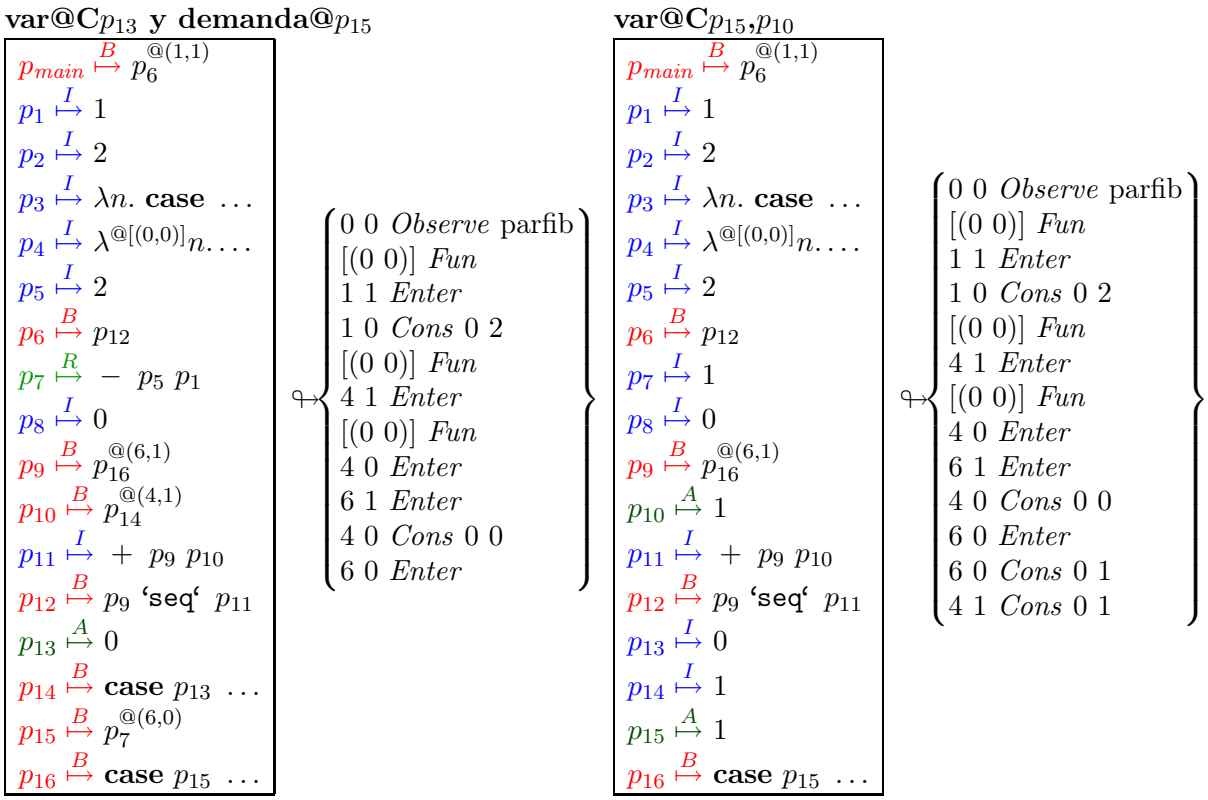
Tras estas reducciones se provoca la demanda de la clausura p_9 que provoca otra observación en el fichero de observaciones. Posteriormente, tras la aplicación de la regla **activación** se activan las clausuras p_8 y p_{16}, que tras un paso de reducción de reglas locales nos llevarán a otra configuración, desde la que se volverán a producir anotaciones de observación en el fichero.

Veamos ambas configuraciones:



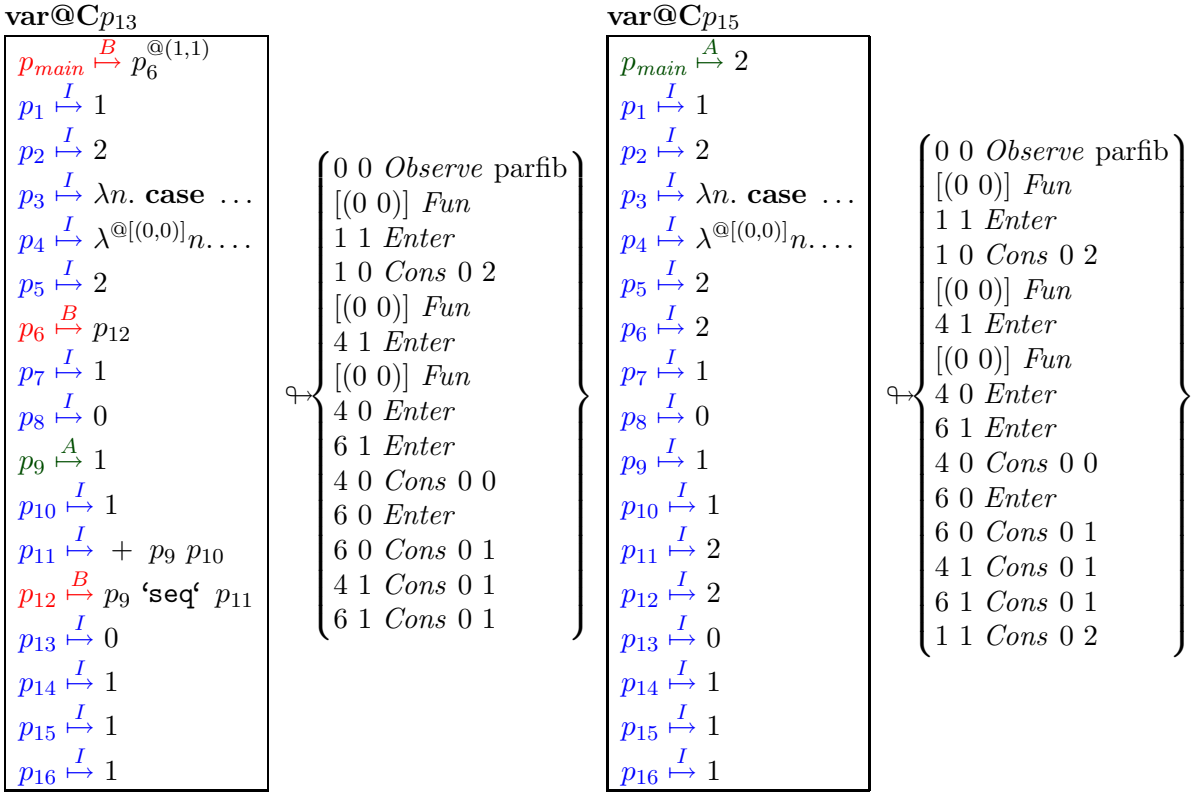
En este caso se vuelve a producir otra demanda en paralelo de las hebras p_{13} y p_{15}. La

evaluación de ambas hebras en paralelo nos lleva a la siguiente configuración:



Al finalizar el cómputo paralelo, es decir, una vez se ha terminado de evaluar la clausura p_{10} (que correspondía con el cómputo en paralelo) y se ha observado su resultado, nos quedamos con

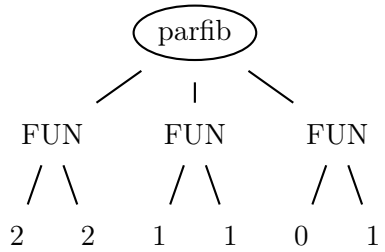
una única hebra en ejecución, la p_{16}, que producirá las observaciones correspondientes.



El cómputo finaliza en este punto. Por tanto, el fichero de observaciones al finalizar el cómputo contendría las siguientes observaciones:

Línea	Observación
0	0 0 *Observe* parfib
1	[(0 0)] *Fun*
2	1 1 *Enter*
3	1 0 *Cons* 0 2
4	[(0 0)] *Fun*
5	4 1 *Enter*
6	[(0 0)] *Fun*
7	4 0 *Enter*
8	6 1 *Enter*
9	4 0 *Cons* 0 0
10	6 0 *Enter*
11	6 0 *Cons* 0 1
12	4 1 *Cons* 0 1
13	6 1 *Cons* 0 1
14	1 1 *Cons* 0 2

Analizando el fichero de igual forma que en los ejemplos del Capítulo 6, se genera el siguiente árbol:



El aplanamiento de ese árbol produce la siguiente observación:

```
-- parfib
   { \ 2 -> 2
   , \ 1 -> 1
   , \ 0 -> 1
   }
```

que es la misma que se produce en la implementación. □

8.4. Semántica formal de Eden

La evaluación de una expresión Eden Core requiere, en general, la creación de varios procesos paralelos. Cada proceso está formado por un conjunto de varias hebras que actúan de forma independiente – cada hebra se encarga de la producción de una salida del proceso– y que comparten los datos del proceso. Recalcamos que la creación de los procesos es explícita, pero la comunicación (y sincronización) es implícita.

El comportamiento local de Eden se define como el conjunto de reglas de las Figuras 8.3 y 8.4. Es interesante resaltar que en Eden no se realiza ninguna distinción entre las hebras ejecutables y activas, por tanto las etiquetas R de las reglas se pueden eliminar o considerar como activas.

Los procesos de Eden poseen su propio *heap* y las ligaduras no son compartidas con otros procesos. Por tanto, es conveniente utilizar un fichero de anotaciones por cada proceso. Un proceso quedará denotado como $\langle id, H \multimap f \rangle$, donde id es el identificador del proceso, H es el *heap* y f es el fichero de anotaciones.

Consideraremos los *streams* como una extensión de Eden. Por eso primero veremos la semántica de Eden sin *streams* y posteriormente añadiremos a dicha semántica la comunicación vía *streams*. De esta manera se presentará la semántica de forma clara y separando los conceptos básicos de las extensiones.

Además de los procesos y la comunicación vía *streams*, Eden posee otras extensiones tales como no-determinismo explícito y canales dinámicos. Ambas extensiones de Eden no serán objeto de esta tesis, pues las reglas semánticas de dichas extensiones no producen ningún efecto sobre las observaciones y sólo llevarían a enturbiar la semántica de la depuración. Aquel lector interesado en la semántica de dichas extensiones puede encontrarla en [Hid04].

8.4.1. Transiciones globales de Eden

A un nivel superior, las transiciones globales de Eden controlan la evolución de los procesos en el sistema, es decir, del conjunto de procesos y sus relaciones. Una transición global posee el siguiente aspecto:

$$\{\overline{\langle id_j, H_j \multimap f_j \rangle}\} \xrightarrow{\diamond} \{\overline{\langle id_j, H'_j \multimap f'_j \rangle}\}$$

$$\mathsf{rch}(H,e) = \mathit{fix}\ (\lambda L\ .\ L \cup \mathit{fv}\ e \cup \bigcup_{p\in L}\{\mathit{fv}\ e' \mid (p \overset{\alpha}{\mapsto} e') \in H\})$$

donde *fix* denota el mínimo punto fijo.

$$\begin{array}{lll}
\mathsf{mo}(id, p \overset{\alpha}{\mapsto} C\ \overline{x_i}) & \overset{\text{def}}{=} & p \overset{\alpha}{\mapsto} C\ \overline{x_i} \\
\mathsf{mo}(id, p \overset{\alpha}{\mapsto} \lambda x.e) & \overset{\text{def}}{=} & p \overset{\alpha}{\mapsto} \lambda x.e \\
\mathsf{mo}(id, p \overset{\alpha}{\mapsto} \lambda^{\overline{[(r_i,s_i)]}} x.e) & \overset{\text{def}}{=} & p \overset{\alpha}{\mapsto} q^{@id +\!\!+ \overline{[(r_i,s_i)]}} \\
 & & q \overset{\alpha}{\mapsto} \lambda x.e \qquad \mathit{fresca}(q) \\
\mathsf{mo}(id, p \overset{\alpha}{\mapsto} e) & \overset{\text{def}}{=} & p \overset{\alpha}{\mapsto} e \qquad e \notin \mathit{whnf}
\end{array}$$

$$\mathsf{nh}(id, e, H) = \{\mathsf{mo}(id, p \overset{I}{\mapsto} e') \mid (p \overset{\alpha}{\mapsto} e') \in H \wedge p \in \mathsf{rch}(H, e)\}$$

Figura 8.7: función nh

$$\begin{array}{ll}
\mathsf{dRch}(H, p) & = \varnothing \text{ si } p \notin \mathit{dom} H \\
\mathsf{dRch}(H, e) & = \bigcup_{p\in \mathit{fv}(e)} \mathsf{dRch}(H, p) \\
\mathsf{dRch}(H \cup p \overset{\alpha}{\mapsto} w, p) & = \mathsf{dRch}(H, w) \\
\mathsf{dRch}(H \cup p \overset{IA}{\mapsto} e, p) & = \{p\} \text{ si } e \notin \mathit{whnf} \\
\mathsf{dRch}(H \cup p \overset{I}{\mapsto} q\#l, p) & = \{p\} \cup \mathsf{dRch}(H, q) \\
\mathsf{dRch}(H \cup p \overset{B}{\mapsto} e, p) & = \{p\} \cup \mathsf{dRch}(H, q) \text{ si } \exists q, l,\ e \in \mathsf{ble}(q) \text{ o } e = q\#l
\end{array}$$

$$\mathsf{nff}(e, H) = \{(p \overset{\alpha}{\mapsto} e') \mid (p \overset{\alpha}{\mapsto} e') \in H \wedge p \in \mathsf{dRch}(H, e)\}$$

Figura 8.8: función nff

donde cada *heap* H_j (asociado con el proceso id_j) se transforma en el *heap* H'_j. Como consecuencia de este proceso pueden crearse nuevos procesos. Además, esta evolución puede producir nuevas anotaciones en el fichero f_j dando como resultado el fichero modificado f'_j.

Las tareas necesarias a nivel global se corresponden con las siguientes: *creación de procesos* (Figura 8.9), *comunicación entre procesos* (Figura 8.10), *manejo de hebras* (Figura 8.11), y *evolución del sistema* (Figura 8.12). En general, estas tareas implican varios pasos simples, donde cada uno de dichos pasos depende a su vez de al menos dos procesos. Con la idea de presentar las transiciones de forma más comprensible, no hemos añadido el fichero de anotaciones en las reglas que no lo modifican, es decir, en aquellas reglas que sólo lo transmiten de la parte izquierda a la parte derecha de las reglas.

Eden realiza las comunicaciones entre procesos de dos formas: enviando un único valor simple por un canal, o a través de los *streams*. En una primera etapa introduciremos la *comunicación de valores simples*. Posteriormente, en la Sección 8.4.3 introduciremos la *comunicación a través de streams*.

Funciones auxiliares

Antes de presentar las reglas globales, definiremos algunas funciones auxiliares que utilizaremos en las reglas.

Función nh**.** Como no existe un *heap* compartido, cuando se crea un nuevo proceso todas las ligaduras necesarias para la evaluación (es decir, las variables libres del cuerpo de la expresión a evaluar en el proceso), deben ser copiadas del *heap* del "padre" al *heap* del "hijo", aunque inactivas, ya que en el proceso en el que se copiará posteriormente este conjunto (de ligaduras) las ligaduras copiadas todavía no han sido demandadas. Para este propósito utilizaremos la función nh (*heap* necesario) definida en la Figura 8.7. Por tanto $\mathsf{nh}(id, e, H)$ recoge todas las ligaduras de H que son alcanzables desde e. Cuando se realiza una copia de valores entre *heaps*, debemos tener en cuenta y modificar apropiadamente las λ-abstracciones que están siendo observadas (las únicas formas normales que pueden tener marcas de observación). Éste es el objetivo de la función $\mathsf{mo}(id, p \mapsto e)$, que sólo se define para formas normales, ya que todas las clausuras que se envían al proceso se mandan en *whnf*. Pero existen varias formas de definir dicha modificación:

1. Tal y como aparece en la Figura 8.7. Donde se modifica la observación para *recordar* de qué proceso provienen, es decir, cuál es su "padre".
2. Dejarlas tal cual vienen, $\mathsf{mo}(id, p \overset{\alpha}{\mapsto} \lambda^{@\overline{[(r_i,s_i)]}}x.e) = p \overset{\alpha}{\mapsto} \lambda^{@\overline{[(r_i,s_i)]}}x.e$. De esta manera no es posible relacionar en el postproceso las marcas de observación de los hijos con el padre. Ahora bien, se pueden obtener de forma independiente las observaciones que se producen en el hijo y en el padre, para lo cual sólo es necesario un sencillo postproceso de los ficheros del hijo y del padre. Esto es lo que realmente sucede en la implementación vista en el Capítulo 7 si tenemos en cuenta que a este nivel, tal y como se comentó en la Sección 7.1, las funciones siempre se encuentran en forma normal.
3. Definirla de la siguiente forma: $\mathsf{mo}(id, p \overset{\alpha}{\mapsto} \lambda^{@\overline{[(r_i,s_i)]}}x.e) = p \overset{\alpha}{\mapsto} \lambda x.e$, es decir, eliminando la observación. De esta forma lo que estamos indicando con las observaciones es que sólo se producen en el proceso en que se crean. Esto no significa que el proceso hijo no pueda observar dicha función. Para realizar dicha observación el programador ha de incluir una nueva marca de observación, dentro del proceso.

Función nff**.** Cuando se comunica un valor, debemos controlar que la expresión a copiar se encuentre en *whnf* (recalcaremos que en la implementación sólo se controla el tipo de la expresión); es más, todas las clausuras alcanzables desde ésta y así recursivamente deben encontrase en *whnf*. La función *needed first free* comprueba dicha propiedad (nff en la Figura 8.8). Esta función retorna las primeras clausuras alcanzables que no se encuentran en *whnf*. Por tanto, un valor e en un *heap* H sólo puede ser enviado a otro proceso (regla de **demanda de la comunicación entre procesos**) si y solo si $\mathsf{nff}(e, H) = \varnothing$. Esta función también es necesaria en la regla **demanda de la creación del proceso**, porque nos devuelve el conjunto de clausuras que es necesario activar para que el proceso se pueda crear impacientemente.

Como puede verse, el desarrollo semántico de la depuración pone de manifiesto más alternativas que la realizada en la implementación. Cualquiera de ellas puede ser considerada como válida a pesar de que los resultados de las observaciones son diferentes: debido a que nos encontramos en un entorno paralelo, las posibilidades se amplían generando un amplio abanico de opciones. En cada implementación se podría optar por una de ellas, pero sería necesario hacer consciente al

(creación de proceso)

$$\text{si } H\ q = \lambda x.e,\ \mathit{frescas}(id', ch_i, ch_o, l'),\ \mathit{renombramiento}(\eta)$$

$$\Big(S, \langle id, H + \{p \overset{\alpha}{\mapsto} q\#l\}\rangle\Big) \xrightarrow{pcu} \left(S,\ \begin{array}{l}\langle id, H + \{p \overset{B}{\mapsto} ch_o, ch_i \overset{A}{\mapsto} l\}\rangle,\\ \langle id', \eta(\mathsf{nh}(id, q, H)) + \{ch_o \overset{A}{\mapsto} \eta(q)\, l', l' \overset{B}{\mapsto} ch_i\}\rangle\end{array}\right)$$

(creación de proceso@)

$$\text{si } H\ q = \lambda^{@[\overline{(r_i,s_i)}]}x.e,\ \mathit{fresh}(id', ch_i, ch_o, l', p'),\ \mathit{renombramiento}(\eta)$$

$$\Big(S, \langle id, H + \{p \overset{\alpha}{\mapsto} q\#l\}\rangle\Big) \xrightarrow{pc@} \left(S,\ \begin{array}{l}\langle id, H + \left\{\begin{array}{l} p \overset{B}{\mapsto} p'^{@(\mathit{length}\ f,1)}, p' \overset{B}{\mapsto} ch_o,\\ ch_i \overset{A}{\mapsto} l^{@(\mathit{length}\ f,0)}\end{array}\right\} \rightarrowtail f \circ \langle \overline{[(r_i\ s_i)]}\ \mathit{Fun}\rangle\rangle,\\ \langle id', \eta(\mathsf{nh}(id, q, H)) + \{ch_o \overset{A}{\mapsto} \eta(q)\, l', l' \overset{B}{\mapsto} ch_i\}\rangle\end{array}\right)$$

Figura 8.9: Eden Core: Creación de proceso

programador de cuál es la opción implementada, para que de esa manera sea capaz de entender las observaciones obtenidas.

En [Hid04] se demostró que la aplicación de una regla simple $\diamond$ a una ligadura particular de un proceso puede habilitar la aplicación de la misma regla a otras ligaduras—en el mismo proceso o incluso en otro—pero no puede desehabilitar la aplicación de $\diamond$ que previamente eran posibles. Las observaciones no intervienen en la activación/desactivación de ligaduras, por tanto, esta propiedad sigue siendo cierta en nuestro caso.

Creación de procesos

Los nuevos procesos se crean cuando se evalúa la expresión #. En este momento se aplica una de las reglas que aparecen en la Figura 8.9. La regla **creación de proceso** maneja la creación de procesos que no están siendo observados, mientras que la regla **creación de proceso@** maneja la creación de procesos que están siendo observados.

Al igual que en la definición de la función nh, en la creación de los procesos se nos plantean varias opciones:

1. Crear los procesos olvidándonos de que el proceso en sí sea una λ-abstracción observada o no. De esta forma la regla **creación de proceso@** sobraría y sería necesario añadir en la regla **creación de proceso** la condición de que el proceso puede ser una λ-abstracción observada. Esta λ-abstracción la tratará adecuadamente la función nh, es decir, se modificaría la observación según se escoja una u otra versión de dicha función.

2. Crear los procesos con la idea de que si el proceso es una λ-abstracción observada debemos observar los canales de entrada y salida, tal y como indica la regla **creación de proceso@**.

Cuando se crea un proceso, la hebra que evalúa la expresión # (en el lado del *padre*) se bloquea en un nuevo canal ch_o, correspondiente con el canal de la nueva hebra del nuevo proceso *s hijo*. De la misma forma, se crea en el proceso hijo una hebra que se encuentra bloqueada en el canal de entrada ch_i; en dicha hebra es donde el padre depositará los datos a comunicar para que el hijo los recoja (comunicación del padre al hijo). Tal como se ha mencionado anteriormente, los

procesos se crean de forma impaciente cuando se encuentran en el nivel superior, es decir, cuando una variable en el *heap* está ligada a una expression #, incluso si dicha ligadura no se encuentra activa (es decir, demandada). Por este motivo se dice que los procesos son *especulativos*, se crean y comienzan a producir resultados aunque no hayan sido demandados.

La única diferencia entre las reglas **creación de proceso** y **creación de proceso@** es que en la segunda debemos anotar con marcas de observación la entrada que se pasa al proceso y la salida que se obtiene del proceso. Nótese que en el fichero del proceso *id* se crea una anotación como si estuviéramos hablando de una λ-abstracción. A partir de ahora, desde el momento en que la evaluación del cuerpo del proceso comience y a medida que vaya siendo demandada la evaluación de sus entradas y se vayan procesando sus salidas, éstas estarán siendo observadas desde el "padre", tal y como dice la regla **β-reducción@**. Pero debido a la modificación que hemos realizado de la observación de la λ-abstracción, cuando la ligadura de ch_o se evalúe, se producirán las observaciones correspondientes en el proceso "hijo", también debido a la regla local **β-reducción@**.

La repetición de ambas reglas nos da lugar a la regla de creación de procesos que se define a continuación:

$$\overset{pc}{\Longrightarrow} = \overset{pc@}{\Longrightarrow} \circ \overset{pcu}{\Longrightarrow}$$

Diferentes posibilidades semánticas

Trabajar a nivel semántico nos lleva a que con un simple cambio en una regla semántica o una definición se produzcan de forma sencilla y natural varias opciones semánticas, todas ellas válidas y a considerar. Como se ha comentado anteriormente, nos podríamos plantear varias opciones semánticas dependiendo de la forma en que se modifican las observaciones y en si observamos o no los datos transmitidos a través de los canales cuando el proceso corresponde con una λ-abstracción observada.

Combinando las 3 posibilidades referentes a la modificación de las observaciones y las 3 posibilidades referentes a la creación de los procesos obtenemos 6 posibles semánticas, que se corresponden con las que se presentan en la siguiente tabla:

	Observar los canales	NO observar los canales
Añadir el padre a las observaciones	▪ se observa en cada proceso lo que evalúa ▪ se observa en el proceso "padre" el cálculo que se manda computar en paralelo en el "hijo" ▪ se mantienen las referencias cruzadas entre los ficheros de observación	▪ se observa en cada proceso sólo lo que evalúa sin observar los cómputos enviados en paralelo ▪ se mantienen las referencias cruzadas entre los ficheros de observación
No modificar las observaciones	▪ se observa en cada proceso lo que evalúa ▪ se observa en el proceso "padre" el cálculo que se manda computar en paralelo en el "hijo" ▪ se pierden las referencias cruzadas entre los ficheros de observación	▪ se observa en cada proceso sólo lo que evalúa sin observar los cómputos enviados en paralelo ▪ se pierden las referencias cruzadas entre los ficheros de observación
Eliminar observaciones	▪ sólo se observa en el proceso "padre" lo que evalúa ▪ se observa en el proceso "padre" el cálculo que se manda computar en paralelo en el "hijo" ▪ no hay observaciones cruzadas	▪ sólo se observa en el proceso "padre" lo que evalúa sin observar los cómputos enviados en paralelo ▪ no hay observaciones cruzadas

La tabla se puede considerar que va de mayor cantidad de observaciones a menor cantidad de observaciones, es decir, al observar los canales y añadir el padre a las observaciones genera más observaciones que no observar los canales y eliminar las observaciones. El extremo superior izquierdo corresponde con el propuesto en este capítulo en el que planteamos la máxima cantidad de observaciones. Mientras que la implementación puede considerarse que sigue la técnica de mantener las observaciones y observar los datos transmitidos por los canales, salvo por el hecho de que las funciones se encuentran directamente en forma normal que no es exactamente lo mismo que las formas normales de la semántica, como ya se ha comentado previamente. De esta forma, en la implementación fue necesario añadir dos anotaciones de observaciones en la función

(comunicación de valores)
si $\mathsf{nff}(w, H_r) = \varnothing$, $renombramiento(\eta)$
$$\Big(S, \langle id_1, H_1 + \{ch \overset{\alpha}{\mapsto} w\}\rangle, \langle id_2, H_2 + \{p \overset{B}{\mapsto} ch\}\rangle\Big) \overset{vCom}{\longrightarrow}$$
$$\overset{vCom}{\longrightarrow} \Big(S, \langle id_1, H_1\rangle, \langle id_2, H_2 + \eta(\mathsf{nh}(id_1, w, H_1)) + \{p \overset{A}{\mapsto} \eta\ w\}\rangle\Big)$$

Figura 8.10: Eden Core: Comunicación entre procesos

de los procesos para realizar el análisis de especulación. Esto se podría considerar equivalente a no observar los canales.

Recordaremos también que la principal ventaja de la implementación consiste en que no es necesario modificar el compilador. Sin embargo, si adoptáramos cualquiera de las otras opciones sería necesario modificar el compilador. Por ejemplo, para implementar la primera opción sería necesario analizar todas las clausuras que van a enviarse al hijo y modificar las marcas en todas aquellas que estén siendo observadas. Con la metodología que generamos nosotros para el análisis de especulación en la Sección 7.3.1, obteníamos los mismos resultados simplemente añadiendo las observaciones en los sitios adecuados, es decir, podíamos conseguir la observación de los cuatro datos que intervienen en un proceso. Como puede observarse, simplemente con añadir las observaciones con cuidado podemos obtener una aproximación muy similar a las propuestas en este capítulo.

Comunicación entre procesos

La regla concerniente con la comunicación de valores se presenta en la Figura 8.10. Cuando se comunica un valor, es obligatorio no sólo copiar dicha ligadura, sino todas las ligaduras correspondientes con las variables alcanzables desde dicho valor—del *heap* del productor al *heap* del consumidor—. La copia sólo puede llevarse a cabo si las expresiones correspondientes a dichas ligaduras se encuentran en *whnf*, es decir, $\mathsf{nff}(w, H_1) = \varnothing$. Renombramos todos los valores y las expresiones en $\mathsf{nh}(id_1, w, H_1)$ utilizando un η–renombramiento para eliminar posibles colisiones con las variables que se encuentren en el *heap* del consumidor. Además, modificamos las observaciones para indicar que provienen del *heap* del consumidor y así posteriormente poder analizar por dónde han pasado.

Definimos la regla de comunicación de la siguiente forma $\overset{Com}{\Longrightarrow} = \overset{vCom}{\Longrightarrow}$. En la Sección 8.4.3 extenderemos $\overset{Com}{\Longrightarrow}$ con la regla de comunicación a través de *streams*.

Manejo de hebras

Las reglas concernientes al manejo de hebras y la evolución del sistema se presentan en la Figura 8.11. El propósito de cada regla es el siguiente:

(desbloqueo de *whnf*) Desbloquear las ligaduras bloqueadas en una ligadura que ha alcanzado *whnf*.

(desactivación de *whnf*) Desactivar las ligaduras que se encuentran en *whnf*.

$$\textbf{(desbloqueo de } \textit{whnf}\textbf{)}$$
$$\text{si } \exists q, e \in \mathsf{ble}(q) \quad \left(S, \langle r, H + \{q \overset{A}{\mapsto} w, p \overset{B}{\mapsto} e\}\rangle\right) \overset{wUnbl}{\longrightarrow} \left(S, \langle r, H + \{q \overset{A}{\mapsto} w, p \overset{A}{\mapsto} e\}\rangle\right)$$

$$\textbf{(desactivación de } \textit{whnf}\textbf{)}$$
$$\left(S, \langle r, H + \{p \overset{A}{\mapsto} w\}\rangle\right) \overset{deact}{\longrightarrow} \left(S, \langle r, H + \{p \overset{I}{\mapsto} w\}\rangle\right)$$

$$\textbf{(demanda de la comunicación de valores)}$$
$$\text{si } p \overset{I}{\mapsto} e \in \mathsf{nff}(w, H)$$
$$\left(S, \langle r, H + \{ch \overset{I}{\mapsto} w, p \overset{I}{\mapsto} e\}\rangle\right) \overset{vComd}{\longrightarrow} \left(S, \langle r, H + \{ch \overset{I}{\mapsto} w, p \overset{A}{\mapsto} e\}\rangle\right)$$

Figura 8.11: Eden Core: Planificación

$$\textbf{(paralela-p)}$$
$$\frac{H^A = \{\overbrace{p^i \overset{A}{\mapsto} e^i}^{n}\} \quad \overbrace{\{H = H^i_U + H^i_M : p_i \overset{A}{\mapsto} e_i \multimap f^{i-1} \longrightarrow H^i_U + K^i \multimap f^i\}}^{n}}{\langle id, H \multimap f^0\rangle \overset{par\text{-}p}{\Longrightarrow} \langle id, \cap_{i=1}^n H^i_U \bigcup \cup_{i=1}^n K^i \multimap f^n\rangle}$$

$$\textbf{(paralela)}$$
$$\frac{\{\langle id_j, H_j \multimap f_j\rangle \overset{par\text{-}p}{\Longrightarrow} \langle id_j, H'_j \multimap f'_j\rangle\}_{\langle id_j, H_j \multimap f_j\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle id_j, H'_j \multimap f'_j\rangle\}_{\langle j, H_j \multimap f_j\rangle \in S}}$$

Figura 8.12: Eden Core: Paralelismo

(demanda de la comunicación de valores) Demandar la evaluación de las variables libres necesarias para la comunicación de los valores entre procesos.

Cuando una ligadura finalmente alcanza su valor, las hebras bloqueadas en ella se desbloquean y la ligadura se desactiva. Después de dicho desbloqueo se cambia el estado de la ligadura en *whnf* a inactivo. Estas reglas disparan las reglas locales **valor** y **demanda**.

La integración y la combinación de las reglas de la Figura 8.11 nos da lugar a la regla:

$$\overset{Unbl}{\Longrightarrow} = \overset{vComd}{\Longrightarrow} \circ \overset{pcd}{\Longrightarrow} \circ \overset{bpc}{\Longrightarrow} \circ \overset{deact}{\Longrightarrow} \circ \overset{wUnbl}{\Longrightarrow}.$$

El número de hebras bloqueadas es finito y no se incrementa debido al desbloqueo y/o la desactivación, como se demostró en [Hid04]. Por tanto, esto garantiza la finitud de la regla *Unbl*.

Reglas de evolución paralela del sistema

Este conjunto de reglas maneja el comportamiento global del sistema. Primero debemos hacer evolucionar cada proceso utilizando las transiciones locales, y posteriormente se hace evolucionar el sistema global. La Figura 8.12 muestra las reglas de la evolución paralela del sistema. La regla **paralela-p** muestra la evolución de un proceso *id* aplicando las reglas de transición local, mientras que la regla **paralela** muestra la evolución global de todos los procesos.

Cada proceso evoluciona internamente realizando transiciones locales (véase la Sección 8.2.1), por tanto, la evolución local de un proceso *id* se determina por el conjunto de hebras activas que

existen en paralelo en H (el *heap* de ligaduras del proceso id) que se permite que progresen. La regla **paralela-p** es exactamente la misma que la regla **paralela** correspondiente de GpH, pero ahora cada proceso posee su *heap* propio H, H^A se corresponde con el conjunto de hebras activas del proceso id y $n = |H^A|$ es el número de hebras activas de id.

Una vez que hemos definido la evolución de los procesos, es necesario definir la evolución global del sistema S. Dicha evolución es manejada por la regla **paralela** que hace evolucionar cada proceso de forma independiente a través de la regla **paralela-p**.

Evolución global del sistema

Después de que cada proceso individual del sistema ha evolucionado a través de las transiciones locales, la evolución global del sistema comienza. Esta evolución viene determinada por la aplicación de las reglas vistas anteriormente. La transición $\overset{sys}{\Longrightarrow}$ maneja todas las comunicaciones pendientes, posteriormente se crean los procesos nuevos y finalmente se realiza el manejo de las hebras y queda definida así:

$$\overset{sys}{\Longrightarrow} = \overset{Unbl}{\Longrightarrow} \circ \overset{pc}{\Longrightarrow} \circ \overset{Com}{\Longrightarrow}$$

El orden es relevante, ya que la comunicación de valores puede habilitar una creación de procesos, pero la creación de procesos nunca habilitará ninguna comunicación pendiente. Por otro lado, el sistema finaliza con las tareas de desbloqueo, desactivación, bloqueo y demanda. Es razonable que se finalice con dichas tareas ya que si no se ha realizado ningún cómputo previamente estas acciones no tienen sentido.

Finalmente, las transiciones del sistema global se representan por $\Longrightarrow$, que se define así:

$$\Longrightarrow = \overset{sys}{\Longrightarrow} \circ \overset{par}{\Longrightarrow} .$$

Por tanto, una computación tiene el siguiente aspecto:

$$S_0 \Longrightarrow S_1 \Longrightarrow \ldots$$

y una condición suficiente para que la computación finalice es que el puntero p_{main} se encuentre inactivo. Sin embargo, existen muchas posibilidades, tales como que no haya ninguna hebra activa en el sistema. Para más detalles véase [HOR07, Hid04].

8.4.2. Ejemplo de evaluación semántica en Eden

Al igual que con la semántica de GpH, creemos conveniente mostrar un ejemplo para comprender mejor el comportamiento de las observaciones. Nos centraremos en los cómputos correspondientes con los pasos de evaluación de las observaciones. Para ello mostraremos la interacción del cómputo paralelo con las marcas de observación.

También nos parece interesante mostrar en el ejemplo las observaciones que se obtienen con las diferentes opciones semánticas comentadas anteriormente. De esta manera veremos de forma clara las ventajas e inconvenientes de cada una de ellas.

Ejemplo 8.3 Para poder comprender mejor las diferencias con GpH, en este ejemplo calcularemos también el número Fibonacci realizando los cómputos en paralelo y observando la función

Fibonacci. El paralelismo lo conseguiremos al evaluar las dos llamadas recursivas de dicha función en paralelo. Nótese que las observaciones también se producirán en paralelo, ya que la llamada recursiva de la función se realiza con la función *parfibO* que produce observaciones. En este ejemplo nos fijaremos únicamente en las observaciones y la evolución paralela.

Partimos, por tanto, de la siguiente expresión en Eden:

```
main = parfib0 2

parfib0 =  observe "parfib" (process parfib)

parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = (parfib0 # (n-1)) + (parfib0 # (n-2))
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
  letrec
    uno   = 1
    dos   = 2

    parfib  = \n. case n of
                    0 -> uno
                    1 -> uno
                    _ -> letrec
                                 n1 = - n uno
                                 n2 = - n dos
                                 nf1 = parfib0 # n1
                                 nf2 = parfib0 # n2
                            in   + nf1 nf2

    parfib0 = parfib@{parfib}
  in parfib0 dos
```

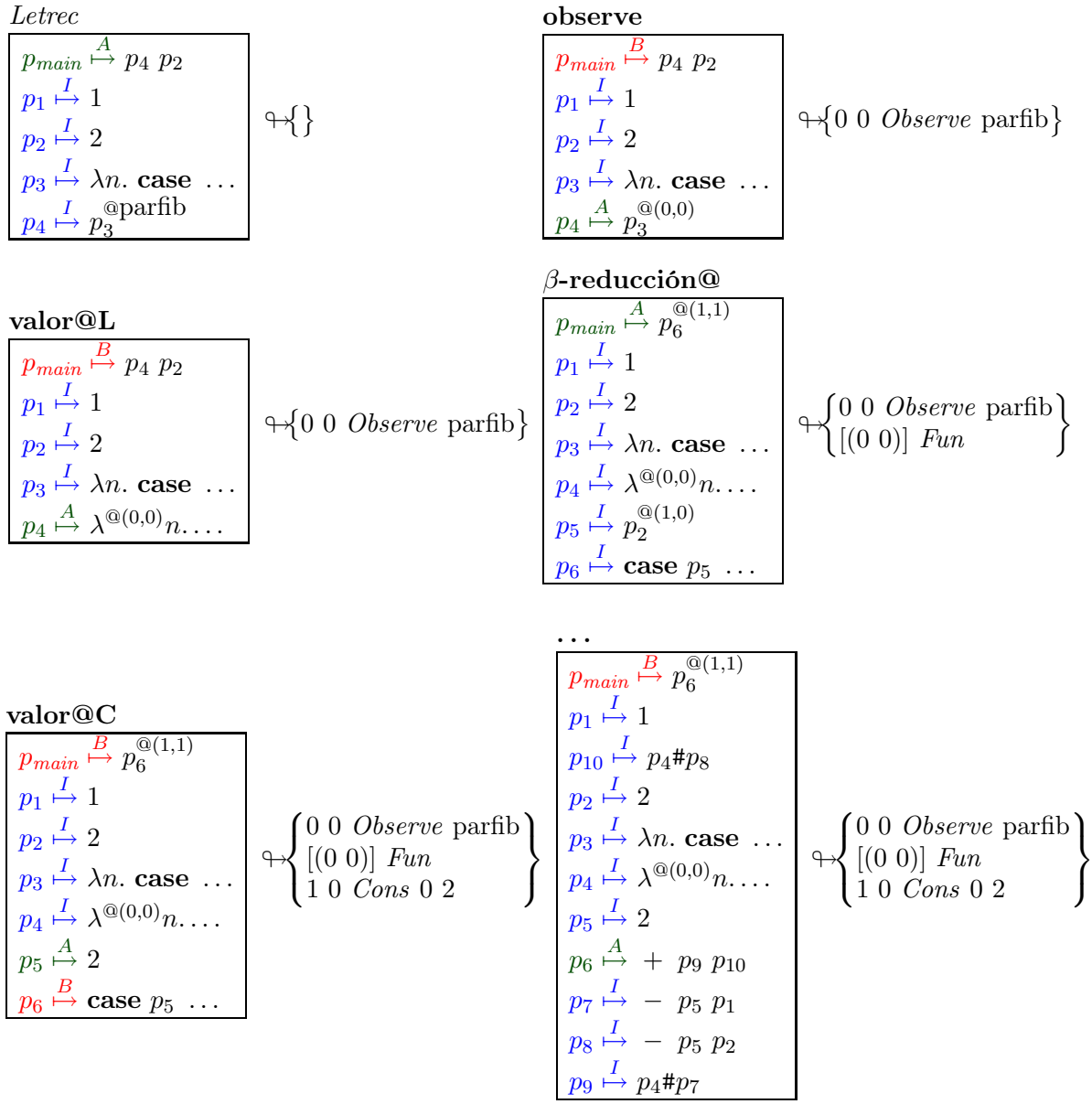
Nótese que el valor de n se corresponde con un entero; recordemos que los enteros se consideran constructores de aridad 0. Pero con el fin de no colapsar las alternativas de la expresión **case** y debido a que la expresión **case** no posee alternativa por defecto, la simularemos con _ que significa que no nos interesa el valor de n.

Al igual que en GpH realizaremos el cálculo del Fibonacci de 2 para forzar a que se produzca la evolución en paralelo. Como en la semántica de Eden no es importante el número de procesadores, ya que se considera que posee todos los que necesita, en este caso veremos que son necesarios tres procesadores, en uno de ellos se evaluará el padre y en los otros dos cada uno de los hijos. Pasemos a ver las reducciones más interesantes en este cómputo. Se parte inicialmente de la configuración siguiente:

$$\boxed{p_{main} \overset{A}{\mapsto} \textbf{letrec } \ldots \textbf{ in } \text{parfibO dos}} \not\rightarrow \{\}$$

Veamos los primeros pasos del cómputo que generan observaciones y la última configuración

antes de que comience la evaluación en paralelo:



La evaluación anterior es similar a la evaluación de GpH. Ahora bien, la evaluación del cómputo es completamente diferente en lo que respecta al paralelismo. De hecho, ahora será necesario crear un *heap* para cada proceso. Es en este momento, cuando tenemos dos procesos vinculados a nivel principal, cuando se van a crear dichas aplicaciones (clausuras p_9 y p_{10}). Veamos las diferentes configuraciones que se producirán para cada una de las posibles semánticas. Mostraremos en este caso los ficheros de observación debajo de cada *heap*.

1. Observar los canales y modificar las anotaciones de observación de los procesos observados añadiéndolas al padre (tal y como se ha presentado en la figuras principales):

main	hijo 1	hijo 2
$p_{main} \overset{B}{\mapsto} p_6^{@(1,1)}$ $p_1 \overset{I}{\mapsto} 1$ $p_2 \overset{I}{\mapsto} 2$ $p_3 \overset{I}{\mapsto} \lambda n.$ **case** n **of** *alts* $p_4 \overset{I}{\mapsto} \lambda^{[(0\ 0)]} n.$**case** n **of** *alts* $p_5 \overset{I}{\mapsto} 2$ $p_6 \overset{A}{\mapsto} +\ p_9\ p_{10}$ $p_7 \overset{I}{\mapsto} -\ p_5\ p_1$ $p_8 \overset{I}{\mapsto} -\ p_5\ p_2$ $p_9 \overset{B}{\mapsto} p_{18}^{@(4,1)}$ $p_{10} \overset{B}{\mapsto} p_{26}^{@(5,1)}$ $p_{18} \overset{B}{\mapsto} ch_{13}$ $p_{26} \overset{B}{\mapsto} ch_{21}$ $ch_{12} \overset{A}{\mapsto} p_7^{@(4,0)}$ $ch_{20} \overset{A}{\mapsto} p_8^{@(5,0)}$	$ch_{13} \overset{A}{\mapsto} p_{15}\ p_{11}$ $p_{11} \overset{B}{\mapsto} ch_{12}$ $p_{14} \overset{I}{\mapsto} \lambda n.$ **case** n **of** *alts* $p_{15} \overset{I}{\mapsto} p_{14}^{@\text{main}[(1\ 0)]}$ $p_{16} \overset{I}{\mapsto} 2$ $p_{17} \overset{I}{\mapsto} 1$	$ch_{21} \overset{A}{\mapsto} p_{23}\ p_{19}$ $p_{19} \overset{B}{\mapsto} ch_{20}$ $p_{22} \overset{I}{\mapsto} \lambda n.$ **case** n **of** *alts* $p_{23} \overset{I}{\mapsto} p_{22}^{@\text{main}[(1\ 0)]}$ $p_{24} \overset{I}{\mapsto} 2$ $p_{25} \overset{I}{\mapsto} 1$
$\left\{\begin{array}{l} 0\ 0\ \textit{Observe}\ \text{parfib} \\ [(0\ 0)]\ \textit{Fun} \\ 1\ 1\ \textit{Enter} \\ 1\ 0\ \textit{Cons}\ 0\ 2 \\ [(0\ 0)]\ \textit{Fun} \\ [(0\ 0)]\ \textit{Fun} \end{array}\right\}$	$\{\}$	$\{\}$

Como se puede observar, tanto los datos que se envían a través de los canales (ch_{12} y ch_{20}) como los que se reciben (p_9 y p_{10}) se encuentran observados. Además, las observaciones de las λ-abstracciones han sido sustituidas por las observaciones que hacen referencia a sus padres (p_{15} y p_{23}). Los canales de salida se corresponden con la aplicación de estas dos nuevas clausuras a los valores que reciben a través de los canales, que se corresponden con las clausuras p_{11} y p_{19} respectivamente. Las dos nuevas anotaciones de observación producidas en el fichero del padre se corresponden con las dos nuevas creaciones de los procesos, de modo que cada anotación de observación hace referencia a una de ellas.

A partir de este momento la evaluación prosigue y los ficheros finales que se obtienen son

los siguientes:

main

Línea	Observación
0	0 0 *Observe* parfib
1	[(0 0)] *Fun*
2	1 1 *Enter*
3	1 0 *Cons* 0 2
4	[(0 0)] *Fun*
5	[(0 0)] *Fun*
6	5 0 *Enter*
7	4 0 *Enter*
8	4 0 *Cons* 0 1
9	5 0 *Cons* 0 0
10	4 1 *Cons* 0 1
11	5 1 *Cons* 0 1
12	1 1 *Cons* 0 2

hijo 1

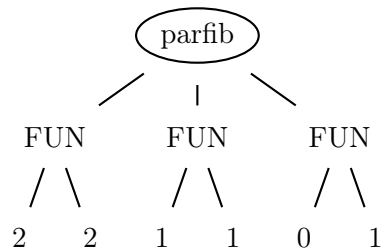
Línea	Observación
0	0 0 *Observe* main[(1 0)]
1	[(0 0)] *Fun*
2	0 1 *Enter*
3	0 0 *Cons* 0 0
4	0 1 *Cons* 0 1

hijo 2

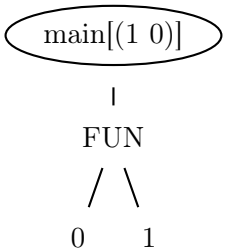
Línea	Observación
0	0 0 *Observe* main[(1 0)]
1	[(0 0)] *Fun*
2	0 1 *Enter*
3	0 0 *Cons* 0 1
4	0 1 *Cons* 0 1

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles para cada fichero:

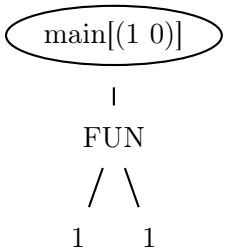
main



hijo 1



hijo 2



El aplanamiento de estos árboles da lugar a las siguientes observaciones que corresponden con las observaciones que se han producido en cada proceso de forma independiente:

main

```
-- parfib
   { \ 2 -> 2
   , \ 1 -> 1
   , \ 0 -> 1
   }
```

hijo 1

```
-- main[(1 0)]
   { \ 0 -> 1
   }
```

hijo 2

```
-- main[(1 0)]
   { \ 1 -> 1
   }
```

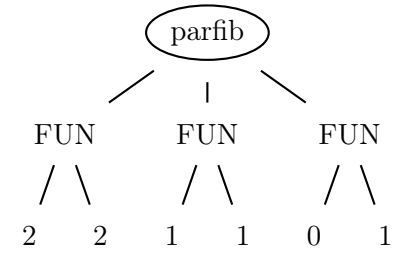
En este caso podemos apreciar que en cada proceso se observan las aplicaciones que han sucedido solamente en ellos. Mientras que en el padre se observa tanto la aplicación que se ha realizado en dicho proceso como las aplicaciones que se han producido en los hijos, ya que se están observando los canales por los que se transmiten los datos.

2. Observar los canales y mantener sin modificación las anotaciones de observación de los procesos:

En este caso, a diferencia del anterior, las clausuras p_{15} y p_{23} no se crearían, pero las clausuras p_{14} y p_{29} estarían vinculadas a la expresión $\lambda^{@(0,0)}n$. **case** n **of** *alts*. Por tanto, los ficheros de observaciones que obtendríamos después del cómputo son los siguientes:

main	hijo 1	hijo 2
Línea Observación 0 0 0 *Observe* parfib 1 [(0 0)] *Fun* 2 1 1 *Enter* 3 1 0 *Cons* 0 2 4 [(0 0)] *Fun* 5 [(0 0)] *Fun* 6 5 0 *Enter* 7 4 0 *Enter* 8 4 0 *Cons* 0 1 9 5 0 *Cons* 0 0 10 4 1 *Cons* 0 1 11 5 1 *Cons* 0 1 12 1 1 *Cons* 0 2	Línea Observación 0 [(0 0)] *Fun* 1 0 1 *Enter* 2 0 0 *Cons* 0 0 3 0 1 *Cons* 0 1	Línea Observación 0 [(0 0)] *Fun* 1 0 1 *Enter* 2 0 0 *Cons* 0 1 3 0 1 *Cons* 0 1

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles:

main	hijo 1	hijo 2
		

Que provocan las siguientes observaciones:

main	hijo 1	hijo 2

`-- parfib`
`{ \ 2 -> 2`
`, \ 1 -> 1`
`, \ 0 -> 1`
`}`
	`--`
	`{ \ 0 -> 1`
	`}`
		`--`
		`{ \ 1 -> 1`
		`}`

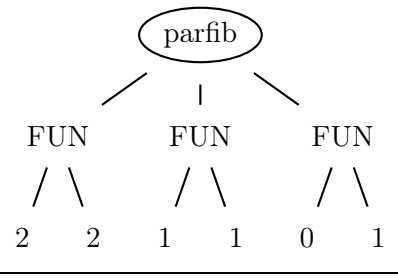
En este caso la diferencia principal con el anterior es que las anotaciones de los hijos no hacen referencia a las del padre.

3. Observar los canales y eliminar las anotaciones de observación de los procesos observados:

La principal diferencia con el apartado anterior es que las clausuras p_{14} y p_{29} estarían vinculadas a la expresión $\lambda n.$ **case** n **of** *alts*. Por tanto, los procesos no realizarían ninguna observación. Es decir, los ficheros de observaciones que obtendríamos después del cómputo son los siguientes:

main		hijo 1	hijo 2
Línea	Observación		
0	0 0 *Observe* parfib		
1	[(0 0)] *Fun*		
2	1 1 *Enter*		
3	1 0 *Cons* 0 2		
4	[(0 0)] *Fun*		
5	[(0 0)] *Fun*	{}	{}
6	5 0 *Enter*		
7	4 0 *Enter*		
8	4 0 *Cons* 0 1		
9	5 0 *Cons* 0 0		
10	4 1 *Cons* 0 1		
11	5 1 *Cons* 0 1		
12	1 1 *Cons* 0 2		

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles:

main	hijo 1	hijo 2
		

Que provocan la siguiente observación:

main	hijo 1	hijo 2

`-- parfib`
`{ \ 2 -> 2`
`, \ 1 -> 1`
`, \ 0 -> 1`
`}`		

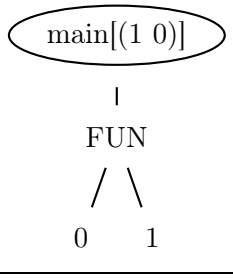
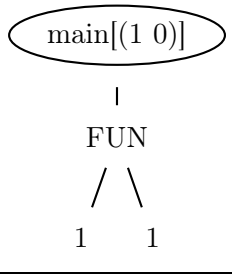
Ahora no se realiza ninguna observación en los hijos, debido a que eliminamos las marcas de observación de las λ-abstracciones que se transmiten a los procesos hijos.

4. No observar los canales y modificar las anotaciones de observación de los procesos observados añadiéndolas el padre:

A diferencia del caso 1, en este caso los canales no se observan, luego no se crean las clausuras p_{18} y p_{26}, se vinculan directamente las clausuras p_9 y p_{10} con sus canales correspondientes ch_{13} y ch_{21} y no se observan las clausuras p_7 y p_8 en los canales ch_{12} y ch_{20}. Por tanto, se producen los siguientes ficheros de observaciones:

main	hijo 1	hijo 2
Línea Observación 0 0 0 *Observe* parfib 1 [(0 0)] *Fun* 2 1 1 *Enter* 3 1 0 *Cons* 0 2 4 1 1 *Cons* 0 2	Línea Observación 0 0 0 *Observe* main[(1 0)] 1 [(0 0)] *Fun* 2 0 1 *Enter* 3 0 0 *Cons* 0 0 4 0 1 *Cons* 0 1	Línea Observación 0 0 0 *Observe* main[(1 0)] 1 [(0 0)] *Fun* 2 0 1 *Enter* 3 0 0 *Cons* 0 1 4 0 1 *Cons* 0 1

Analizando cada fichero independientemente y siguiendo el mismo proceso que todos los ejemplos anteriores se obtienen los siguientes árboles:

main	hijo 1	hijo 2
		

En este caso, se produce una observación de la aplicación de la λ-abstracción en cada fichero de observaciones, la correspondiente al cómputo que se ha producido en el proceso correspondiente. Por tanto, se produce la siguiente observación:

main	hijo 1	hijo 2

`-- parfib`
`{ \ 2 -> 2`
`}`
	`-- main[(1 0)]`
	`{ \ 0 -> 1`
	`}`
		`-- main[(1 0)]`
		`{ \ 1 -> 1`
		`}`

Este caso es similar al caso 1, salvo que ahora como no estamos observando los datos transmitidos por los canales, en cada proceso sólo se observan las aplicaciones que se han realizado en dicho proceso.

5. No observar los canales y mantener sin modificación las anotaciones de observación de los procesos:

   La diferencia con el caso anterior es que ahora tampoco se crean las clausuras p_{15} y p_{23} y que las clausuras p_{14} y p_{29} estarían vinculadas a la expresión $\lambda^{@(0,0)}n.\mathbf{case}\ n\ \mathbf{of}\ alts$. Por tanto, los ficheros de observaciones que obtendríamos después del cómputo son los

siguientes:

main	hijo 1	hijo 2
Línea Observación 0 0 0 *Observe* parfib 1 [(0 0)] *Fun* 2 1 1 *Enter* 3 1 0 *Cons* 0 2 12 1 1 *Cons* 0 2	Línea Observación 0 [(0 0)] *Fun* 1 0 1 *Enter* 2 0 0 *Cons* 0 0 3 0 1 *Cons* 0 1	Línea Observación 0 [(0 0)] *Fun* 1 0 1 *Enter* 2 0 0 *Cons* 0 1 3 0 1 *Cons* 0 1

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles:

main	hijo 1	hijo 2
parfib | FUN / \ 2 2	FUN / \ 0 1	FUN / \ 1 1

Que provocan las siguientes observaciones:

main	hijo 1	hijo 2

`-- parfib`
`{ \ 2 -> 2`
`}`
	`--`
	`{ \ 0 -> 1`
	`}`
		`--`
		`{ \ 1 -> 1`
		`}`

La diferencia con el apartado anterior consiste en que ahora los procesos hijos no producen una marca que haga referencia a la observación del padre, por tanto, no se sabe a qué corresponden dichas observaciones.

6. No observar los canales y eliminar las anotaciones de observación de los procesos observados (tal y como hace la implementación):

   La principal diferencia con el apartado anterior es que las clausuras p_{14} y p_{29} estarían vinculadas a la expresión $\lambda n.$ **case** n **of** *alts*. Por tanto, los procesos no realizarían ninguna observación, es decir, los ficheros de observaciones que obtendríamos después del cómputo son los siguientes:

main	hijo 1	hijo 2
Línea Observación 0 0 0 *Observe* parfib 1 [(0 0)] *Fun* 2 1 1 *Enter* 3 1 0 *Cons* 0 2 4 1 1 *Cons* 0 2	{}	{}

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles:

main	hijo 1	hijo 2
		

Que provocan la siguiente observación:

main	hijo 1	hijo 2

`-- parfib`
`{ \ 2 -> 2`
`}`		

En este caso, como estamos eliminando las observaciones de las λ-abstracciones, no se produce ninguna observación en los procesos hijo. □

Como queda patente en estos ejemplos, las diferentes opciones semánticas dan lugar a diferentes observaciones. Consideramos menos interesantes los casos 2 y 5, ya que en ellos se pierde la idea de la procedencia de las observaciones. Sin embargo, los otros cuatro casos son bastante interesantes.

8.4.3. Manejo de *streams* en Eden

A través de los canales de comunicación de datos, entre los procesos vistos en la sección anterior sólo se pueden mandar valores simples. Nótese que se puede mandar una lista definida por el usuario a través de dichos canales, pero la lista tiene que encontrarse previamente evaluada antes de ser transmitida a través del canal. Para mejorar las comunicaciones, Eden permite a los procesos comunicarse a través de *streams*, es decir, listas de elementos cuyos elementos se envían tan pronto como éstos alcanzan la *forma normal*, pero la ventaja radica en que la lista no tiene que encontrarse completamente evaluada, pues los elementos de la lista se envían uno a uno.

Para manejar los *streams* debemos introducir una nueva regla local, dos reglas de comunicación y una regla global de demanda de comunicación (véase la Figura 8.13). Téngase en cuenta que ninguna de estas reglas modifican el fichero de anotaciones. Entraremos a explicarlas en detalle a continuación.

Regla local demanda *stream*. Cuando un canal se encuentra vinculado con un *stream* y su cabeza aún no ha sido evaluada, se crea una demanda ficticia sobre el canal para conseguir la evaluación del canal de forma impaciente. Esto se debe a que en Eden tanto la creación de procesos como la comunicación de los datos se realiza de forma especulativa.

(demanda-*stream*)

si $e \notin whnf$

$$H + \{p_1 \overset{IAB}{\mapsto} e\} : ch \overset{A}{\mapsto} [p_1 : p_2] \longrightarrow H + \{p_1 \overset{AAB}{\mapsto} e, ch \overset{B}{\mapsto} [p_1 : p_2]\}$$

(comunicación *stream*-vacío)

$$(S, \langle r, H_r + \{ch \overset{\alpha}{\mapsto} \texttt{nil}\}\rangle, \langle s, H_s + \{p \overset{B}{\mapsto} ch\}\rangle) \overset{eCom}{\longrightarrow} (S, \langle r, H_r\rangle, \langle s, H_s + \{p \overset{A}{\mapsto} \texttt{nil}\}\rangle)$$

(comunicación de la cabeza del *stream*)

si $(\mathsf{nff}(w, H_r + \{ch \overset{\alpha}{\mapsto} [p_1 : p_2], p_1 \overset{I}{\mapsto} w\}) = \varnothing)$, $frescas(q_1, q_2)$, $renombramiento(\eta)$

$$(S, \langle r, H_r + \{ch \overset{\alpha}{\mapsto} [p_1 : p_2], p_1 \overset{I}{\mapsto} w\}\rangle, \langle s, H_s + \{p \overset{B}{\mapsto} ch\}\rangle \overset{sCom}{\longrightarrow}$$
$$\overset{sCom}{\longrightarrow} (S, \langle r, H_r + \{ch \overset{A}{\mapsto} p_2, p_1 \overset{I}{\mapsto} w\}\rangle, \langle s, H_s + \eta(\mathsf{nh}(s, w, H_r)) + \{p \overset{A}{\mapsto} [q_1 : q_2], q_1 \overset{A}{\mapsto} \eta(w), q_2 \overset{B}{\mapsto} ch\}\rangle)$$

(demanda de la comunicación de la cabeza del *stream*)

if $p \overset{I}{\mapsto} e \in \mathsf{nff}(w, H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2]\})$

$$(S, \langle r, H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2]\}\rangle) \overset{hsComd}{\Longrightarrow} (S, \langle r, H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2], p \overset{A}{\mapsto} e\}\rangle)$$

Figura 8.13: Eden Core: Reglas de manejo de *streams*

Regla global comunicación *stream*-vacío. Cuando el valor a ser comunicado se corresponde con el final del *stream* (`nil`), el valor se comunica y el *stream* se cierra, por tanto no se puede comunicar ningún otro valor a través de dicho canal.

Regla global comunicación de la cabeza del *stream*. En el caso de los *streams*, cuando el valor en cabeza del *stream* se encuentra evaluado y, por tanto, preparado para ser comunicado (es decir, dicho valor y todas las variables libres alcanzables desde él se encuentran en *whnf*), dicho valor se pasa del productor al consumidor renombrando las variables con unas frescas. Nótese que la interacción con las observaciones se realiza a través de la función nh. la cual maneja las observaciones y las modifica adecuadamente.

Regla global demanda de la comunicación de la cabeza del *stream*. Cuando un canal se encuentra ligado a un *stream* y su valor en cabeza no está completamente evaluado, se crea demanda sobre las primeras clausuras alcanzables que no se encuentren en *whnf*.

La comunicación de un valor entre procesos—o un valor simple o la cabeza de un *stream*—puede dar lugar a más comunicaciones. Estas reglas **demanda de la comunicación del valor**, **comunicación *stream*-vacío** y **comunicación de la cabeza del *stream*** deben ser aplicadas repetidamente hasta que no se pueda comunicar ningún otro valor. Por tanto se define

$$\overset{Com}{\longrightarrow} = \overset{vCom}{\Longrightarrow} \circ \overset{eCom}{\Longrightarrow} \circ \overset{sCom}{\Longrightarrow}$$

y $\overset{Com}{\Longrightarrow}$ de la manera habitual.

8.4.4. Ejemplo de comunicación vía *streams* en Eden

Nótese que la comunicación que se producía en el Ejemplo 8.3 era completa ya que era un único valor el que transmitíamos. Pero aunque hubiéramos transmitido un par de valores, como éstos se transmiten en forma normal, su comunicación se hubiera realizado en un único paso de evaluación. Para ello hubiera sido necesario reducir a *whnf* todas las clausuras alcanzables. Los *streams* solucionan este problema de comunicación, ya que transmiten los valores a través del *stream* de uno en uno. Veamos un ejemplo en que se comuniquen los datos entre los procesos vía *streams* y en que exista especulación, de esta manera nos podremos fijar en los detalles relativos a la transmisión vía *streams* y a la especulación.

Ejemplo 8.4 En este caso la especulación surge debido a que el padre necesita solamente el segundo valor de la lista de datos que produce el hijo. Mientras, el hijo produce la lista infinita de números enteros a partir de un `n` inicial. Partimos, por tanto, de la siguiente expresión en Eden:

```
main = elemI 2 (lDesdeNO # 7)

lDesdeNO = observe "lDesdeN" lDesdeN

lDesdeN :: Int -> [Int]
lDesdeN n = n:lDesdeN (n+1)

elemI :: Int -> [a] -> a
elemI 1 (x:xs) = x
elemI n (x:xs) = elemI (n-1) xs
```

que en nuestro lenguaje tras el proceso de normalización se convierte en la siguiente expresión de partida e_0:

```
letrec
  uno   = 1
  dos   = 2
  siete = 7

  elemI  = \n\ys. case n of
                    1 -> case ys of
                           Cons x xs -> x
                    _ -> case ys of
                           Cons x xs -> letrec
                                           n1 = - n uno
                                           elemN1 = elemI n1
                                        in  elemN1 xs

  lDesdeN  = \n. letrec
                   n1 = + n uno
                   lns = lDesdeN n1
                 in (n:lns)

  lDesdeNO = lDesdeN@{lDesdeN}

  instEden = lDesdeNO # siete
  elemDos = elemI dos
```

```
in elemDos instEden
```

La reducción de la expresión **letrec** nos dará lugar a la siguiente configuración:

$$
\begin{array}{l}
p_{main} \overset{A}{\mapsto} p_8\ p_7 \\
p_1 \overset{I}{\mapsto} 1 \\
p_2 \overset{I}{\mapsto} 2 \\
p_3 \overset{I}{\mapsto} 7 \\
p_4 \overset{I}{\mapsto} \lambda n.\ \lambda ys.\ \textbf{case}\ n\ \textbf{of}\ alts \\
p_5 \overset{I}{\mapsto} \lambda n.\ \textbf{letrec} \\
\qquad\qquad\qquad n1 = +\ n\ p_1 \\
\qquad\qquad\qquad lns = p_5\ n1 \\
\qquad\qquad \textbf{in}\ (n : lns) \\
p_6 \overset{I}{\mapsto} p_5^{@\mathrm{lDesdeN}} \\
p_7 \overset{I}{\mapsto} p_6 \# p_3 \\
p_8 \overset{I}{\mapsto} p_4\ p_2
\end{array}
\quad \rightarrowtail \{\}
$$

En este momento, debido a la regla **demanda de la creación de proceso** se demandará la evaluación de la hebra p_6, necesaria para la creación del proceso. Tras su reducción se creará el nuevo proceso dando lugar a la siguiente configuración:

main	hijo
$ch_{10} \overset{A}{\mapsto} p_3^{@[(1\ 0)]}$ $p_{16} \overset{B}{\mapsto} ch_{11}$ $p_{main} \overset{B}{\mapsto} p_8\ p_7$ $p_1 \overset{I}{\mapsto} 1$ $p_2 \overset{I}{\mapsto} 2$ $p_3 \overset{I}{\mapsto} 7$ $p_4 \overset{I}{\mapsto} \lambda n.\ \lambda ys.\ \textbf{case}\ n\ \ldots$ $p_5 \overset{I}{\mapsto} \lambda n.\ \textbf{letrec}\ \ldots\ \textbf{in}\ (n : lns)$ $p_6 \overset{A}{\mapsto} \lambda^{@[(0\ 0)]} n.\ \textbf{case}\ n\ \ldots$ $p_7 \overset{B}{\mapsto} p_{16}^{@[(1\ 1)]}$ $p_8 \overset{A}{\mapsto} \lambda ys.\ \textbf{case}\ p_2\ \ldots$	$ch_{11} \overset{A}{\mapsto} p_{13}\ p_9$ $p_9 \overset{B}{\mapsto} ch_{10}$ $p_{12} \overset{I}{\mapsto} \lambda n.\ \textbf{letrec}\ \ldots\ \textbf{in}\ (n : lns)$ $p_{13} \overset{I}{\mapsto} p_{12}^{@\mathrm{main}[(1\ 0)]}$ $p_{14} \overset{I}{\mapsto} 1$ $p_{15} \overset{I}{\mapsto} \lambda n.\ \textbf{letrec}\ \ldots\ \textbf{in}\ (n : lns)$
$\{0\ 0\ \mathit{Observe}\ \mathrm{lDesdeN}\}$	$\{\}$

A partir de este momento comienza el cómputo en paralelo. El hijo comenzará a producir resultados y éstos serán demandados por las reglas **demanda del *stream*** y **demanda de la cabeza del *stream***, y transmitidos, por la regla **comunicación de la cabeza del *stream***. Veremos las primeras aplicaciones de estas reglas. La configuración que se presenta a continuación surge debido a la aplicación de la regla **demanda del *stream*** que activa la cabeza del *stream*:

main	hijo
$p_{main} \overset{B}{\mapsto}$ **case** p_7 **of** *alts* $p_1 \overset{I}{\mapsto} 1$ $p_{16} \overset{B}{\mapsto} ch_{11}$ $p_2 \overset{I}{\mapsto} 2$ $p_3 \overset{I}{\mapsto} 7$ $p_4 \overset{I}{\mapsto} \lambda n.\ \lambda ys.$ **case** n ... $p_5 \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_6 \overset{I}{\mapsto} \lambda^{@(0,0)} n.$ **letrec** ... **in** $(n : lns)$ $p_7 \overset{B}{\mapsto} p_{16}^{@(1,1)}$ $p_8 \overset{I}{\mapsto} \lambda ys.$ **case** p_2 ...	$ch_{11} \overset{B}{\mapsto} (p_{21} : p_{22})$ $p_{12} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{13} \overset{I}{\mapsto} \lambda^{@(0,0)} n.$ **letrec** ... **in** $(n : lns)$ $p_{14} \overset{I}{\mapsto} 1$ $p_{15} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{17} \overset{I}{\mapsto} p_9^{@(1,0)}$ $p_{18} \overset{I}{\mapsto} (p_{17} : p_{20})$ $p_{19} \overset{I}{\mapsto} +\ p_{17}\ p_{14}$ $p_{20} \overset{I}{\mapsto} p_{15}\ p_{19}$ $p_{21} \overset{A}{\mapsto} p_{17}^{@(3,0)}$ $p_{22} \overset{I}{\mapsto} p_{20}^{@(3,1)}$ $p_9 \overset{I}{\mapsto} 7$
$\left\{\begin{array}{l} 0\ 0\ \textit{Observe}\ \text{lDesdeN} \\ [(0\ 0)]\ \textit{Fun} \\ 1\ 0\ \textit{Cons}\ 0\ 7 \end{array}\right\}$	$\left\{\begin{array}{l} 0\ 0\ \textit{Observe}\ \text{main}[(1\ 0)] \\ [(0\ 0)]\ \textit{Fun} \\ 1\ 1\ \textit{Enter} \\ 1\ 1\ \textit{Cons}\ 2\ : \end{array}\right\}$

Antes de enviar el dato a través del canal, el cómputo prosigue hasta que las clausuras alcanzables desde la clausura p_6 se encuentren en *whnf*. Es en ese instante cuando se producirá la comunicación debido a la regla **comunicación de la cabeza del *stream***. A continuación mostraremos la configuración de partida y la configuración resultante tras el envío del dato a

través del *stream*:

main	hijo
$p_{main} \overset{B}{\mapsto}$ **case** p_7 **of** *alts* $p_1 \overset{I}{\mapsto} 1$ $p_{16} \overset{B}{\mapsto} ch_{11}$ $p_2 \overset{I}{\mapsto} 2$ $p_3 \overset{I}{\mapsto} 7$ $p_4 \overset{I}{\mapsto} \lambda n.\ \lambda ys.$ **case** n ... $p_5 \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_6 \overset{I}{\mapsto} \lambda^{@[(0,0)]} n.$**letrec** ... **in** $(n : lns)$ $p_7 \overset{B}{\mapsto} p_{16}^{@(1,1)}$ $p_8 \overset{I}{\mapsto} \lambda ys.$ **case** p_2 ...	$ch_{11} \overset{B}{\mapsto} (p_{21} : p_{22})$ $p_9 \overset{I}{\mapsto} 7$ $p_{12} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{13} \overset{I}{\mapsto} \lambda^{@[(0,0)]} n.$**letrec** ... **in** $(n : lns)$ $p_{14} \overset{I}{\mapsto} 1$ $p_{15} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{17} \overset{I}{\mapsto} 7$ $p_{18} \overset{I}{\mapsto} (p_{17} : p_{20})$ $p_{19} \overset{I}{\mapsto} +\ p_{17}\ p_{14}$ $p_{20} \overset{I}{\mapsto} p_{15}\ p_{19}$ $p_{21} \overset{A}{\mapsto} 7$ $p_{22} \overset{I}{\mapsto} p_{20}^{@(3,1)}$
0 0 *Observe* lDesdeN [(0 0)] *Fun* 1 0 *Cons* 0 7	0 0 *Observe* main[(1 0)] [(0 0)] *Fun* 1 1 *Enter* 1 1 *Cons* 2 : 3 0 *Enter* 1 0 *Cons* 0 7 3 0 *Cons* 0 7

La configuración tras la aplicación de la regla **comunicación de la cabeza del *stream*** es la

siguiente:

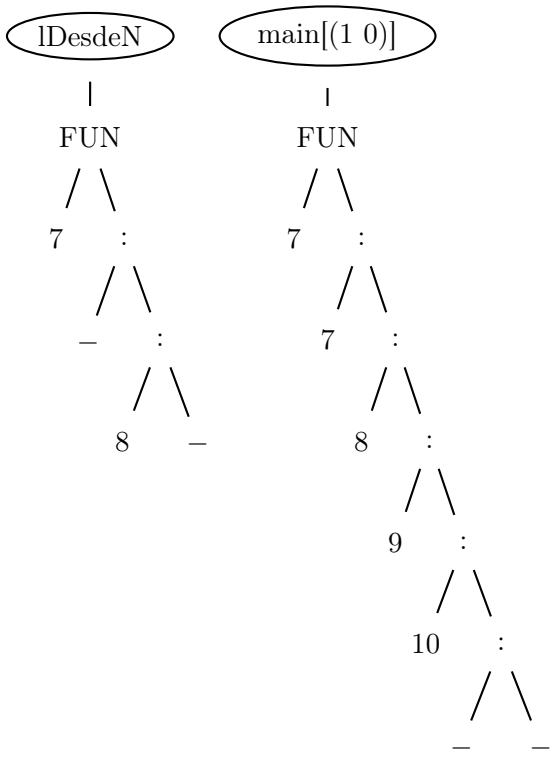
main	hijo
$p_{main} \overset{B}{\mapsto}$ **case** p_7 **of** *alts* $p_1 \overset{I}{\mapsto} 1$ $p_{16} \overset{A}{\mapsto} (p_{23} : p_{24})$ $p_2 \overset{I}{\mapsto} 2$ $p_{23} \overset{A}{\mapsto} 7$ $p_{24} \overset{B}{\mapsto} ch_{11}$ $p_3 \overset{I}{\mapsto} 7$ $p_4 \overset{I}{\mapsto} \lambda n.\ \lambda ys.$ **case** n ... $p_5 \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_6 \overset{I}{\mapsto} \lambda^{@(0,0)} n.$**letrec** ... **in** $(n : lns)$ $p_7 \overset{B}{\mapsto} p_{16}^{@(1,1)}$ $p_8 \overset{I}{\mapsto} \lambda ys.$ **case** p_2 ...	$ch_{11} \overset{A}{\mapsto} p_{22}$ $p_9 \overset{I}{\mapsto} 7$ $p_{12} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{13} \overset{I}{\mapsto} \lambda^{@(3,0)} n.$**letrec** ... **in** $(n : lns)$ $p_{14} \overset{I}{\mapsto} 1$ $p_{15} \overset{I}{\mapsto} \lambda n.$ **letrec** ... **in** $(n : lns)$ $p_{17} \overset{I}{\mapsto} 7$ $p_{18} \overset{I}{\mapsto} (p_{17} : p_{20})$ $p_{19} \overset{I}{\mapsto} +\ p_{17}\ p_{14}$ $p_{20} \overset{I}{\mapsto} p_{15}\ p_{19}$ $p_{21} \overset{A}{\mapsto} 7$ $p_{22} \overset{I}{\mapsto} p_{20}^{@(6,1)}$
0 0 *Observe* lDesdeN [(0 0)] *Fun* 1 0 *Cons* 0 7	0 0 *Observe* main[(1 0)] [(0 0)] *Fun* 1 1 *Enter* 1 1 *Cons* 2 : 3 0 *Enter* 1 0 *Cons* 0 7 3 0 *Cons* 0 7

A partir de este momento la evaluación prosigue y los ficheros finales que se obtienen dependen de la cantidad de cómputo que realice el hijo, es decir, de la especulación. El hijo finaliza una vez que el padre obtiene todos los datos necesarios (en ejecución envía una orden a cada proceso para que éstos terminen). Supondremos que el hijo produce un par de datos adicionales. Entonces los

ficheros serían los siguientes:

main	hijo
Línea Observación 0 0 0 *Observe* lDesdeN 1 [(0 0)] *Fun* 2 1 0 *Cons* 0 7 3 1 1 *Cons* 2 : 4 3 1 *Cons* 2 : 5 4 0 *Cons* 0 8	Línea Observación 0 0 0 *Observe* main[(1 0)] 1 [(0 0)] *Fun* 2 1 1 *Enter* 3 1 1 *Cons* 2 : 4 3 0 *Enter* 5 1 0 *Cons* 0 7 6 3 0 *Cons* 0 7 7 3 1 *Enter* 8 3 1 *Cons* 2 : 9 8 0 *Enter* 10 8 0 *Cons* 0 8 11 8 1 *Enter* 12 8 1 *Cons* 2 : 13 12 0 *Enter* 14 12 0 *Cons* 0 9 15 12 1 *Enter* 16 12 1 *Cons* 2 : 17 16 0 *Enter* 18 16 0 *Cons* 0 10 19 16 1 *Enter* 20 16 1 *Cons* 2 :

Analizando cada fichero independientemente y siguiendo el mismo proceso que en todos los ejemplos anteriores se obtienen los siguientes árboles:



El aplanamiento de dichos árboles producirá las siguientes observaciones:

```
-- lDesdeN                          -- main[(1 0)]
   { \ 7 -> _ : 8 : _                  { \ 7 -> 7 : 8 : 9 : 10 : _ : _
   }                                   }
```

En estas observaciones queda patente la especulación en dos sentidos:

- el padre no ha necesitado para su cómputo el primer valor enviado por el hijo. Este tipo de especulación es difícil de eliminar en un programa,
- el hijo ha producido dos elementos más que el padre no ha consumido y se encuentra evaluando el siguiente, es decir, los elementos 9 y 10 de la lista producidos por el hijo no han sido consumidos por el padre. En este caso la especulación se debe a que el hijo produce constantemente datos de tal forma que el padre no es capaz de procesarlos a esa velocidad o, como en este caso, ni siquiera los necesita. Este tipo de especulación se podría tratar de evitar en ciertas situaciones.

□

8.5. Corrección y equivalencia

Ahora que intuitivamente tenemos una idea de que la extensión semántica es correcta. Creemos conveniente demostrar formalmente que las marcas de observación no cambian el significado de una expresión, es decir, si se evalúa una expresión con marcas de observación y la equivalente

sin marcas debemos obtener como resultado el mismo valor final. Además, debemos probar que la evaluación de una expresión con observaciones no demanda la evaluación de nuevas expresiones que no se demanden en la evaluación de la expresión original sin marcas. Si esto pasara, podría suceder que no terminara la evaluación de la expresión observada. Esto sucede, por ejemplo, si se demanda la evaluación de una expresión que no se demanda con la expresión sin observar, y esta nueva expresión demandada no alcanza *whnf*, debido a que no termina. Entonces las marcas de observación cambiarían la semántica de la expresión original.

La primera diferencia de nuestra semántica con respecto a la original consiste en las marcas de observación. Es por este motivo que para comparar ambas semánticas debemos definir una función que *elimine* las marcas de observación. Esta función consiste en una extensión de la función que definimos en la Sección 6.2.2 y transforma cualquier expresión con marcas de observación GpH-Eden Core$^{@}$ a una expresión sin marcas de observación.

Definición 8.3 La función que elimina las marcas de observación es la siguiente:

$$\mathsf{R} \ : \text{GpH-Eden Core}^{@} \longrightarrow \text{GpH-Eden Core}$$

Se define de forma recursiva, todos los casos son triviales excepto las expresiones observadas:

$$\begin{array}{lcl}
\mathsf{R} \ :: \ \text{GpH} - \text{Eden Core}^{@} \rightarrow \text{GpH} - \text{Eden Core} & & \\
\mathsf{R}\ x & \stackrel{\text{def}}{=} & x \\
\mathsf{R}\ \lambda\ x.e & \stackrel{\text{def}}{=} & \lambda\ x.\mathsf{R}\ e \\
\mathsf{R}\ x\ y & \stackrel{\text{def}}{=} & x\ y \\
\mathsf{R}\ (\textbf{letrec}\ \overline{x_i\ =\ e_i}\ \textbf{in}\ e) & \stackrel{\text{def}}{=} & \textbf{letrec}\ \overline{x_i\ =\ \mathsf{RB}\ e_i}\ \textbf{in}\ \mathsf{R}\ e \\
\mathsf{R}\ (C\ \overline{x_i}) & \stackrel{\text{def}}{=} & C\ \overline{x_i} \\
\mathsf{R}\ (\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow e_i}) & \stackrel{\text{def}}{=} & \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{y_{ij}} \rightarrow \mathsf{R}\ e_i} \\
\mathsf{R}\ prim & \stackrel{\text{def}}{=} & prim \\
\mathsf{R}\ op\ \overline{x_i} & \stackrel{\text{def}}{=} & op\ \overline{x_i} \\
\mathsf{R}\ x\ \texttt{`seq`}\ y & \stackrel{\text{def}}{=} & x\ \texttt{`seq`}\ y \\
\mathsf{R}\ x\ \texttt{`par`}\ y & \stackrel{\text{def}}{=} & x\ \texttt{`par`}\ y \\
\mathsf{R}\ x\ \texttt{\#}\ y & \stackrel{\text{def}}{=} & x\ \texttt{\#}\ y
\end{array}$$

$$\begin{array}{lcl}
\mathsf{RB} \ :: \ \text{GpH} - \text{Eden Core}^{@}\ \text{Ligadura} \rightarrow \text{GpH} - \text{Eden Core Ligadura} & & \\
\mathsf{RB}\ e & \stackrel{\text{def}}{=} & \mathsf{R}\ e \\
\mathsf{RB}\ x^{@str} & \stackrel{\text{def}}{=} & x \\
\mathsf{RB}\ p^{@(r,s)} & \stackrel{\text{def}}{=} & p \\
\mathsf{RB}\ \lambda^{@\overline{[(r_i,s_i)]}}x.e & \stackrel{\text{def}}{=} & \lambda x.\mathsf{R}\ e
\end{array}$$

Esta función se extiende de forma natural para trabajar con *heaps*. Básicamente, $\mathsf{R}\ H$ se define como $\{p \mapsto \mathsf{RB}\ e \mid (p \mapsto e) \in H\}$.

□

Una vez que ya hemos definido la función que elimina las observaciones podemos pasar directamente a definir la equivalencia entre *heaps*. En este caso, a diferencia de la Sección 6.2.2, no

es necesario definir la aplicación de un *heap* a una expresión, ya que la definición de equivalencia es sustancialmente diferente.

La principal dificultad para definir dicha equivalencia, al igual que en la definición de equivalencia vista en la Sección 6.2.2, es que las reglas que manejan las observaciones introducen nuevos punteros que llamaremos "de paso". Nuestras nuevas reglas **valor@C** y **β-reduction@** son las que generan dichos punteros. Estos punteros apuntan a la expresión original, pero están marcados con marcas de observación que recuerdan que las clausuras originales se encuentran bajo observación. Por tanto, debemos demostrar que las expresiones que aparecen en ambos formalismos son *esencialmente* las mismas, por lo que no se puede considerar como equivalencia matemática que mantendría las propiedades reflexiva, transitiva y simétrica, que esta no las mantiene. Esta equivalencia entre dos *heaps* $H \equiv H'$ debe mantener la propiedad de que $\forall (p \overset{\alpha}{\mapsto} e) \in H$ la ligadura equivalente correspondiente $(p' \overset{\alpha}{\mapsto} e') \in H'$ posea el mismo valor que la expresión e en H. Lo que significa que debemos obtener las mismas expresiones siguiendo los punteros. En este caso, la definición de equivalencia que generaremos será menos abstracta que la de la Sección 6.2.2. Como los cómputos en esta semántica son de paso corto, es más sencillo definir la equivalencia que se encargue de ignorar dichos punteros. Sin embargo ahora será necesario tener en cuenta que las clausuras dentro del *heap* se encuentran etiquetadas con su estado.

Resumiendo, en la semántica con observaciones pueden existir punteros intermedios hasta alcanzar el valor equivalente en la semántica sin observaciones. Estos punteros pueden encontrarse anotados con marcas de observación. El análisis referente a las etiquetas de las ligaduras es el siguiente: cuando la ligadura correspondiente en la semántica sin observaciones se encuentre en estado *activo*, en la semántica con observaciones uno de los punteros intermedios se encontrará en estado *activo*, los punteros anteriores se encontrarán en estado *bloqueado*, ya que estarán esperando el resultado, y los punteros posteriores se encontrarán en estado *inactivo*. Cuando la ligadura sin observaciones se encuentre en estado *inactivo* o *bloqueado*, todos los punteros intermedios en la semántica con observaciones se encontrarán en estado *inactivo* o *bloqueado* respectivamente. Finalmente, cuando la ligadura sin observaciones se encuentre en estado *ejecutable*, el primer o el último puntero de los punteros intermedios en la semántica con observaciones se encontrará en estado *ejecutable*. En caso de que sea el primero el que se encuentre en estado *ejecutable*, los demás se encontrarán en estado *inactivo* y si es el último los anteriores estarán *bloqueados* esperando a que este termine su cómputo. La definición que tiene en cuenta estos detalles es la siguiente:

Definición 8.4 Sean H, H' dos *heaps*. Diremos que:

- $H \equiv H'$ si $\exists \eta -$ renombramiento, $\forall (p \overset{\beta}{\mapsto} e) \in H$ entonces $\exists n \in \mathbb{N}, \{\overline{q_i \overset{\alpha_i}{\mapsto} q_{i+1}}^{n-1}, q_n \overset{\alpha_n}{\mapsto} e'\}$ siendo $\eta\ p = q_1$ y donde se cumplen las siguientes condiciones:
  - $rp\ e \equiv_{\mathsf{R}} rp\ e'$
  - y se cumple una de las siguientes condiciones:
    - si $\beta = A$ entonces $\exists k \in \{1 \dots n\}$, $\alpha_k = A$, $\forall j \in \{1 \dots k-1\}$, $\alpha_j = B$ y $\forall j \in \{k+1 \dots n\}$, $\alpha_j = I$.
    - si $\beta = I$ entonces $\forall j \in \{1 \dots n\}$, $\alpha_j = I$.
    - si $\beta = B$ entonces $\forall j \in \{1 \dots n\}$, $\alpha_j = B$.
    - si $\beta = R$ entonces se cumple una de las siguientes condiciones:

$\diamond$ $\alpha_1 = R$ y $\forall j \in \{2 \ldots n\}$, $\alpha_j = I$
$\diamond$ $\alpha_n = R$ y $\forall j \in \{1 \ldots n-1\}$, $\alpha_j = B$

- $H \equiv_{\mathsf{R}} H'$ si $H \equiv \mathsf{R}\ H'$

□

Debemos demostrar la equivalencia de la evaluación de una expresión con marcas de observación y la correspondiente sin marcas de observación en GpH y en Eden. El siguiente teorema establece la equivalencia en GpH. Recalcaremos que GpH sólo tiene un único *heap*:

Teorema 8.1 *Para toda $e \in$ GpH Core$^{@}$ entonces:*

$$\{p_{main} \overset{A}{\mapsto} \mathsf{R}\ e\} \not\hookrightarrow \langle\ \rangle \Longrightarrow H \not\hookrightarrow \langle\ \rangle \textit{ sii } \{p_{main} \overset{A}{\mapsto} e\} \not\hookrightarrow \langle\ \rangle \Longrightarrow H' \not\hookrightarrow f$$
$$y\ H \equiv_{\mathsf{R}} H'$$

Una idea intuitiva de la demostración de este teorema sería la siguiente: la diferencia principal con respecto a las transiciones locales es que GpH tiene los operadores `‘seq‘` y `‘par‘` pero dichos operadores no se encuentran afectados por las observaciones. Así que con respecto a las reglas locales son equivalentes. Por otro lado, las reglas globales **desactivación** y **activación** no modifican el fichero de observaciones ni las marcas de observación, por tanto, sólo tenemos que analizar el comportamiento de la regla **paralela**: En la unión de los *heaps* $\cup_{i=1}^{n} K^i$ podrían aparecer conflictos con los punteros observables. No obstante, está demostrado que en GpH no se produce ninguna interferencia. Esto significa que si en dos *heaps* K^j y K^k existe una ligadura para un puntero p entonces tenemos el mismo valor en el resto de *heaps* es decir, si $(p \overset{\alpha}{\mapsto} e_j) \in K^j$ y $(p \overset{\beta}{\mapsto} e_k) \in K^k$ entonces $\alpha = \beta$ y $e_j = e_k$. Con respecto a las marcas de observación tampoco se produce ningún cambio. Supongamos por un momento que las clausuras con marcas de observación de dos *heaps* son diferentes; $(p \overset{\alpha}{\mapsto} q^{@(n_1,n_2)}) \in K^j$ y $(p \overset{\alpha}{\mapsto} q^{@(n'_1,n'_2)}) \in K^k$. Es decir, $(p \overset{A}{\mapsto} q^{@str}) \in H$, así que $(p \overset{A}{\mapsto} q^{@str}) \in H^A$. Entonces, la ligadura $p \overset{A}{\mapsto} q^{@str}$ sería una de las que se ha visto involucrada en la regla **paralela**. Supongamos que su índice es i_0. Como es una hebra activa no se modifica por las otras hebras involucradas en dicha regla, por tanto $(p \overset{A}{\mapsto} q^{@str}) \in H_l$ para todo $l \neq i_0$. Por lo que necesariamente tenemos que $j = k = i_0$, y entonces $n_1 = n'_1$ y $n_2 = n'_2$.

En este caso, debido a que no se eliminan las clausuras bajo observación del *heap*, no es necesario realizar ninguna modificación en la semántica para demostrar este teorema de forma sencilla. La demostración de este teorema sigue las ideas de la prueba del Teorema 6.1. Por lo que, previamente demostraremos una proposición más general y así obtendremos como corolario el teorema.

Proposición 8.1 *Sea H un heap sin marcas de observación y $H^{@}$ un heap cualquiera tales que $H \equiv_{\mathsf{R}} H^{@}$ y sea f un fichero, entonces:*

$$H \Longrightarrow K \textit{ sii } H^{@} \not\hookrightarrow f \Longrightarrow K^{@} \not\hookrightarrow f'$$
$$y\ K \equiv_{\mathsf{R}} K^{@}$$

Nótese que esta proposición parece más simple que la Proposición 6.4 ya que no aparece directamente en la proposición que los *heaps* tienen que ser equivalentes para cualquier otra expresión. Esto se cumple directamente debido a la nueva definición de equivalencia entre *heaps*, que tiene en cuenta todas las clausuras y a que la semántica actual es de paso corto.

Demostración 9 Aquí solo presentaremos un esquema de la prueba. El lector interesado encontrará la prueba completa en el apéndice de esta tesis.

La demostración se realizará por inducción sobre la derivación. Para ello veremos que que las reglas locales con observación generan configuraciones equivalentes a las reglas sin observación. En algún caso será necesario realizar varios pasos en las reglas que contienen las observaciones, mientras que sólo será necesario un único paso para el cómputo en el *heap* equivalente sin observaciones.

El siguiente teorema no sólo demuestra que las marcas de observación no cambian el comportamiento del sistema en Eden. También prueba que las marcas de observación no producen nuevos procesos.

Teorema 8.2 *Para toda* $e \in$ *Eden Core*$^{@}$ *entonces:*

$$\{\langle main, \{p_{main} \overset{A}{\mapsto} \mathsf{R}\ e\} \multimap \langle\,\rangle\rangle\} \Longrightarrow \{\overline{\langle r, H_r \multimap \langle\,\rangle\rangle}\} \text{ sii } \{\langle main, \{p_{main} \overset{A}{\mapsto} e\} \multimap \langle\,\rangle\rangle\} \Longrightarrow \{\overline{\langle r, H'_r\rangle \multimap f_r}\}$$

$$y\ H_r \equiv_{\mathsf{R}} H'_r$$

La demostración de este teorema requiere dos etapas. Primero debemos demostrar que las reglas locales en Eden mantienen la equivalencia. Afortunadamente las reglas locales en Eden son un subconjunto de la reglas locales de GpH, por tanto, esta equivalencia se mantiene ya que el teorema anterior es correcto y la regla **paralela-p** de Eden es equivalente a la regla **paralela** de GpH. Por lo que, asumiendo que las reglas locales son correctas, debemos demostrar que las transiciones $\overset{sys}{\Longrightarrow}$ mantienen la equivalencia. Para ello será necesario ver que las reglas de comunicación mantienen la equivalencia y que la creación de procesos, reglas **creación de proceso** y **creación de proceso@**, que son las únicas reglas que tiene un comportamiento diferente si se trabaja con las observaciones, genera *heaps* equivalentes. Por tanto, la transición $\overset{sys}{\Longrightarrow}$ mantiene la equivalencia.

La demostración de este teorema sigue las ideas de la prueba del Teorema 6.1. Por lo que, previamente demostraremos una proposición más general y así obtendremos como corolario el teorema.

Proposición 8.2 *Sea* $\overline{\langle id_i, H_i\rangle}^n$ *un sistema sin marcas de observación y* $\overline{\langle id_i, H'_i \multimap f_i\rangle}^n$ *un sistema cualquiera tales que* $\overline{H_i \equiv_{\mathsf{R}} H'_i}^n$ *entonces:*

$$\overline{\langle id_i, H_i\rangle}^n \Longrightarrow \overline{\langle id_i, K_i\rangle}^m \text{ sii } \overline{\langle id_i, H'_i \multimap f_i\rangle}^n \Longrightarrow \overline{\langle id_i, K'_i \multimap f'_i\rangle}^m$$

$$y\ \overline{K_i \equiv_{\mathsf{R}} K'_i}^m$$

La diferencia con el caso anterior es que ahora tenemos que tener en cuenta que existe un *heap* por cada proceso, por tanto, para probar que un sistema es equivalente a otro hay que demostrar que todos los *heaps* son equivalentes.

Demostración 10 En este caso la demostración se realizará en tres pasos:

1. Veremos que las reglas de evaluación local partiendo de *heaps* equivalentes nos llevan a *heaps* equivalentes.

2. Veremos que la función nh genera *heaps* equivalentes cuando se parten de *heaps* equivalentes, por lo que demostraremos que las reglas de transmisión de datos, es decir, **(comunicación de valores)**, **(comunicación *stream*-vacío)** y **(comunicación de la cabeza del *stream*)** mantiene la relación de equivalencia.

3. Veremos que la regla **creación de proceso** y la regla **creación de proceso@** son equivalentes bajo la equivalencia $\equiv_{\mathsf{R}}$.

La última cuestión importante a demostrar es que de alguna manera la traza que observamos, tanto en GpH como en Eden, se corresponde con lo que hemos anotado como observable y se ha evaluado, es decir, con lo que intuitivamente queremos observar. El siguiente teorema, que es esencialmente el mismo que el Teorema 6.2 establece dicha propiedad:

Teorema 8.3

1. *Si para la evaluación de una expresión es necesario evaluar una variable observada entonces se produce en el fichero del proceso que evalúa dicha ligadura la observación de su resultado.*

2. *Si en el fichero de un proceso hay una marca de observación, ésta se ha producido ya que dicho proceso ha demandado y evaluado la ligadura observada.*

Demostración 11

1. La demostración de esta propiedad es algo tan sencillo como que las únicas reglas que evalúan dichas expresiones son las que aparecen en la Figura 8.4 y que dichas reglas generan en el fichero una marca de observación.

2. La demostración, al igual que en el caso anterior, se realiza de forma trivial indicando que las únicas reglas que añaden observaciones en el fichero son las relativas a la reducción de la marcas observadas.

Por tanto, queda demostrado tanto que la semántica es correcta como que las marcas de observación se realizan de manera adecuada.

Capítulo 9

Conclusiones y trabajo futuro

Después de todos estos capítulos, si tuviéramos que destacar una contribución práctica de esta tesis sería que hemos desarrollado la primera herramienta útil para la depuración de dos lenguajes funcionales paralelos: GpH y Eden. Además, basándonos en ella hemos desarrollado una herramienta que permite analizar la especulación en el lenguaje Eden. Ahora bien, esto que se acaba de resaltar sólo se corresponde con la parte más visible y "maquinera" del trabajo desarrollado en esta tesis.

Sin embargo, consideramos el Capítulo 8 como el más importante y el que de alguna manera resume el trabajo global realizado en esta tesis. En dicho capítulo presentamos la semántica del depurador en los lenguajes paralelos GpH y Eden y demostramos ciertas propiedades de equivalencia entre la evaluación de expresiones marcadas con observaciones y las equivalentes sin marcas. Por otro lado, el trabajo más duro, pero más gratificante, fue el de generar la semántica del depurador en secuencial. Ya que a partir de dicha semántica no resultó excesivamente difícil la inclusión en la semántica paralela. Simplemente fue necesario adaptarse a los distintos niveles que conlleva una semántica de un lenguaje paralelo.

Recapitulemos el trabajo realizado en esta tesis. Primeramente hicimos un estudio exhaustivo de los depuradores que existen en los lenguajes funcionales perezosos, más concretamente en Haskell. A partir de ese momento nos decidimos por extender el depurador Hood al entorno paralelo, ya que lo consideramos como uno de los depuradores más sencillos de manejar. En este punto vimos que dicho depurador carecía de semántica y que incluso el propio autor recomendaba darle una semántica. Es aquí donde comenzó nuestra tarea: por un lado, creemos conveniente formalizar, y por otro lado, el mismo autor de Hood instaba a que dicho trabajo se hiciera. Aunque normalmente el comportamiento del depurador es intuitivo para un programador funcional la tarea de dar semántica a dicho depurador no fue una tarea sencilla, sin embargo, nos llevó a una comprensión mayor del funcionamiento de dicho depurador porque, entre otras cosas, hubo que estudiar el código fuente de dicha librería. En particular comprobamos que el comportamiento de los valores primitivos no es el que uno espera, y que se muestran aunque no se usen en el cómputo, por lo que decidimos incluirlos en nuestro estudio formal. En este punto, una de las conclusiones a las que llegamos es que el desarrollo formal de las aplicaciones hace que se posea un mayor conocimiento de su comportamiento.

No obstante, nuestro estudio del depurador secuencial Hood no sólo quedó en un estudio formal de dicho depurador, sino que también propusimos varias opciones para incluirlo a nivel de la máquina abstracta STG. Entre ellas se estudiaron dos implementaciones de dicho depurador: la

librería original de Hood y la implementación que posee el intérprete Hugs. Además, se propuso una nueva versión que a nivel de la máquina abstracta STG mantenía la compartición de las clausuras que Hood no posee. Sin embargo, la modificación de un compilador como el GHC (que implementa la máquina abstracta STG) es una tarea dura y difícil de mantener, debido a la evolución constante de dicho compilador. Por este motivo estas modificaciones no llegaron a implementarse más que en un prototipo que se utilizó para desarrollar los ejemplos de forma automática. La relevancia de este estudio consiste en ver que a nivel de la máquina abstracta STG se puede incluir la depuración con ciertas mejoras que no se pueden lograr cuando ésta se implementa con una librería externa al compilador. He aquí una línea abierta de trabajo futuro.

A partir de este momento, nos planteamos alcanzar nuestra meta, es decir, conseguir el primer depurador en programación funcional paralela perezosa. Esta idea se bifurcó por dos ramas de investigación. Por un lado, dar una semántica a las observaciones en paralelo y, por otro lado, implementar la depuración en el entorno paralelo. Ambas tareas se fueron intercalando en el tiempo, y a medida que estábamos tratando de dar semántica a las observaciones, en paralelo estábamos implementando la depuración en entornos paralelos. Por una parte, somos conscientes de que antes de empezar a trabajar hay que saber qué se quiere hacer, es decir, la semántica debe preceder a la implementación. El trabajo semántico nos condujo a plantearnos muchas dudas, sobre todo en el lenguaje Eden, que permite la creación de procesos. La principal duda radica en qué significa observar un proceso. Llegamos a la conclusión de que observar un proceso corresponde con la observación de cualquiera de los cuatro datos involucrados en su cómputo, es decir, los datos que el padre envía al hijo, los datos que el hijo utiliza para sus cómputos, los datos que el hijo envía al padre y los datos que el padre utiliza para sus cómputos. Así pues, teniendo en cuenta dichos datos desarrollamos un esquema general para la observación de procesos. Ahora bien, por otra parte, también somos conscientes de que si deseamos que nuestro mecanismo de depuración se utilice como herramienta para depurar los lenguajes funcionales paralelos GpH y Eden no podíamos realizar modificaciones en el compilador, ya que estas modificaciones normalmente suelen quedarse obsoletas cuando se realiza una nueva versión de dicho compilador y, por tanto, todo el trabajo realizado para la modificación de la versión anterior, a veces, no resulta útil para la modificación de la nueva versión. Por esta razón a nivel semántico han surgido varias opciones y a nivel de implementación sólo ha surgido una única versión. A medida que estudiábamos las opciones semánticas que se nos planteaban, nos dábamos cuenta de cuáles de ellas conllevarían una modificación del compilador e inicialmente desechábamos su implementación. Como en el caso secuencial esta línea de investigación queda abierta; la inclusión de nuestras modificaciones no depende únicamente de nosotros, sino que si se la quiere dar una continuidad hay que involucrar al resto de la comunidad que desarrolla el compilador GHC. La conclusión a la que nos llevó este estudio es que, al igual que antes, el estudio formal de las cosas nos hace comprenderlas mejor y abrir muchas opciones. Por su parte, la implementación nos suele centrar en una única línea de desarrollo. Creemos también relevante dar una semántica de la implementación, ya que ésta ayuda a comprender mejor el comportamiento de dicha implementación.

La librería de observaciones implementada consiste en una modificación de la librería original. Esta modificación soporta tanto las observaciones de la ejecución secuencial de un programa Haskell, como las observaciones de los programas de GpH y Eden. Esta librería puede considerarse como el primer depurador en el entorno funcional paralelo-perezoso. Su uso puede ayudar a localizar errores de programación, e implica una pequeña modificación en el código del programa

a observar anotando las clausuras que el programador considera susceptibles de que no se evalúen correctamente, es decir, implica una pequeña intuición del programador que le guíe hacia el código incorrecto. Un uso más sistemático de ella consistiría en partir de la función principal e ir anotando todas las funciones a las que ésta llama. Una vez encontrada la función errónea habría que repetir el proceso. Como se puede observar, es una herramienta versátil, que permite distintas metodologías de uso.

Uno de los usos que se nos ocurrió para esta herramienta consiste en utilizarla para realizar un análisis de especulación en el lenguaje Eden. La modificación de dicha librería fue mínima y la herramienta que obtuvimos permite analizar la especulación de un programa en Eden en cualquier punto del cómputo. El estudio realizado para la observación de los procesos en Eden fue reutilizado para el análisis de especulación. Es más, los esqueletos de Eden fueron redefinidos para incluir el estudio de la especulación. De esta forma el manejo de la herramienta se realiza de forma simple: sólo es necesario utilizar el esqueleto que realiza el análisis en lugar del esqueleto original y finalmente procesar los ficheros de observaciones que se obtiene por una herramienta que los analiza. Esta herramienta no sólo es útil para corregir los programas, sino que también es útil para comprender mejor el comportamiento de los programas paralelos en Eden. Por tanto, debe unirse a otras herramientas habituales en Eden, como es Paradise [HPR00]. Además, el uso de la librería de observaciones para realizar el análisis de especulación nos introduce en la idea de que dicha librería puede ser utilizada para el desarrollo de otros análisis, aquí hay otra línea abierta de investigación.

Finalmente, como conclusión consideramos relevante el trabajo presentado en esta tesis por los siguientes motivos:

- Presenta varias posibilidades de depuración que podrían ser incorporadas en la máquina STG.
- Formaliza el comportamiento de la librería Hood y demuestra que las observaciones no interfieren con el resultado obtenido, únicamente producen observaciones como efecto lateral.
- Presenta la primera implementación de un depurador en el entorno de programación funcional paralela-perezosa. Este depurador funciona tanto en GpH como en Eden.
- Formaliza el comportamiento de la versión paralela de dicho depurador, demostrando que, al igual que antes, las observaciones no interfieren con el resultado obtenido, únicamente producen observaciones como efecto lateral.
- Utiliza la librería para el desarrollo del análisis de especulación en el lenguaje Eden.
- Presenta un método de desarrollo de depuradores.

Como trabajo posterior debemos recordar que esta tesis deja abierta varias líneas de investigación futuras.

Una línea de investigación futura abierta que nos parece bastante viable es la utilización de la librería de observaciones para el desarrollo de diversos análisis. Debido a que la librería realiza anotaciones en tiempo de ejecución sobre un fichero independiente, resulta sencillo postprocesar dichas anotaciones, de manera que marcando las clausuras adecuadas para que se generen dichas anotaciones, se puedan generar varios análisis que partan de dicho fichero. Por ejemplo, podría realizarse de forma muy simple un análisis de especulación en GpH o un análisis de duplicación

de trabajo en Eden. Con respecto a a este último, recordemos que cuando en Eden se crea un proceso, es posible que se copien en el hijo clausuras que estaban sin evaluar en el padre. En dicho caso, si tanto el padre como el hijo necesitan su valor, ambos deberán realizar el mismo cómputo, por lo que se duplicará trabajo. Dicha circunstancia puede ser deseable en determinadas ocasiones, mientras que en otros casos puede resultar negativa. Para analizar si una misma expresión ha sido evaluada tanto por el padre como por el hijo, sólo haría falta *observar* la expresión correspondiente, comprobando posteriormente en cuántos procesadores se han generado anotaciones de observación relativas a dicha expresión.

Otra tarea pendiente con amplias expectativas consiste en la formalización de otros depuradores. Los depuradores en programación funcional perezosa se han ido desarrollando sin tener en cuenta la formalización y la demostración de su corrección. Por este motivo creemos que este área puede ser un campo prometedor en el que falta mucho por hacer.

Por otro lado, la extensión paralela del depurador Hood que hemos realizado sólo se ha probado que funciona en GpH y Eden. Resultaría muy interesante comprobar si este depurador funciona sobre otras extensiones paralelas de Haskell y en caso de que no funcionara extenderlo de manera adecuada a dichas extensiones.

Por último, otras tareas pendientes que deja abierta esta tesis son la implementación del depurador Hood a nivel de la máquina STG (es decir, modificando el compilador GHC) y la implementación de las diferentes versiones paralelas propuestas en el Capítulo 8. De esta última existe un intérprete que ha sido utilizado para generar los ejemplos de forma automática. Sin embargo, la implementación real conllevaría la modificación del compilador de Eden (compilador basado en GHC). Consideramos que ambas líneas de trabajo no son tan prometedoras, ya que la modificación del compilador GHC consiste en una tarea ardua y difícil de llevar a cabo. Además, como ya se ha comentado, dicho compilador sigue activo y constantemente se actualiza su código. Por tanto, mantener las modificaciones activas en dicho compilador conllevaría mucho trabajo. Este es el motivo principal por el que la línea de investigación propuesta por el depurador HsDebug parece que ha dejado de ser una línea de investigación abierta. Además, cualquiera de las observaciones que se obtienen con las versiones propuestas en el Capítulo 8 se puede conseguir con la librería actual anotando las clausuras de la forma adecuada y postprocesando los ficheros de observaciones.

Apéndice A

Demostración de las proposiciones

En este apéndice presentaremos la demostración formal de los teoremas, proposiciones y propiedades. Hemos considerado interesante separar las demostraciones formales para hacer más legible la tesis. Es por ese motivo que en los capítulos previos hemos introducido un esquema de demostración en cada teorema, pero no hemos mostrado todos los detalles.

Ahora bien, como es lógico, consideramos interesante realizar dichas demostraciones, ya que parte importante de esta tesis se corresponde con el desarrollo formal. Es más, el método expuesto en esta tesis podría ser aplicado a otros entornos y, por tanto, las demostraciones podrían ser reutilizadas con las modificaciones adecuadas. Por estos motivos consideramos suficiente el esquema presentado tras cada teorema, pero añadimos en este apéndice las demostraciones para que el lector interesado las pueda analizar.

A.1. Demostración de las proposiciones del Capítulo 6

En esta sección presentaremos dos demostraciones:

- la demostración de equivalencia entre la semántica de una expresión observada y la equivalente sin observar,
- la demostración de equivalencia entre la máquina abstracta derivada de la semántica y la evaluación semántica de la misma expresión.

Utilizaremos la notación $q^{@}$ y $\lambda^{@}$ para denotar un puntero y una λ-abstracción que pueden estar observados, respectivamente. También utilizaremos la abreviatura *h.i.* que significará *hipótesis de inducción.*

Demostración 12 (Proposición 6.4) La demostración se realiza por inducción sobre las reglas.

Como comentamos en el esquema de la demostración, no consideraremos el fichero de observaciones, ya que no es vinculante para el cómputo de las expresiones.

Para cada apartado será necesario demostrar estas tres tesis:

$$\begin{array}{rl} H^{@} : e^{@} \Downarrow_{A,C} K^{@} : w^{@} & \textbf{T1} \\ K : w \equiv_{\mathsf{R}} K^{@} : w^{@} & \textbf{T2} \\ K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@} & \textbf{T3} \end{array}$$

indicaremos que se ha probado cada una de ellas con la siguiente marca (Ti).

Demostración de la primera implicación: $\Rightarrow$

Lam **o** *Cons*

Hipótesis:

H1 $H : e \equiv_{\mathsf{R}} H^{@} : e^{@}$

H2 $H : e \Downarrow_{A,C} K : w$

H3 Por la regla *Lam* o *Cons*:

a) $K : w = H : e$

b) $e = \lambda x.e'$ o $e = C\ \overline{p_i}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1 y H3(b):
$e^{@} = \lambda^{@} x.e'^{@}$ o $e^{@} = C\ \overline{q_i}$

P2 Por la regla *Lam*, *Lam*@ o *Cons*:
$H^{@} : e^{@} \Downarrow_{A,C} H^{@} : e^{@}$ (T1)

P3 Por H3(a) y H1:
$K : w = H : e \equiv_{\mathsf{R}} H^{@} : e^{@}$ (T2)

P4 Por H3(a) y H4:
$K : e' = H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$ (T3)

Letrec

Hipótesis:

H1 $H : \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \equiv_{\mathsf{R}} H^{@} : e_0^{@}$

H2 $H : \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \Downarrow_{A,C} K : w$

H3 Por la regla *Letrec*:

a) $H \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \Downarrow_{A,C} K : w$

b) $\overline{p_i}$ son frescos con respecto a H, A, C y $\mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0^{@} = \mathbf{letrec}\ \overline{x_i = e_i^{@}}\ \mathbf{in}\ e^{@}$

P2 Por H1, H3(b), H4, P1 y Definición 6.3:

a) $H \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \equiv_{\mathsf{R}} H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}}$ si $\overline{q_i}$ son frescas

b) $H \cup [\overline{p_i \mapsto \hat{e}_i}] : e' \equiv_{\mathsf{R}} H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : e'^{@}$ si $\overline{q_i}$ son frescas

P3 Por P2 y h.i. sobre H3(a):

a) $H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}} \Downarrow_{A,C} K^{@} : w^{@}$

b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)
c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P4 Por P3(a) y la regla *Letrec*:
$H^{@} : \mathbf{letrec}\ \overline{x_i = e_i^{@}}\ \mathbf{in}\ e^{@} \Downarrow_{A,C} K^{@} : w^{@}$ (T1)

Case

Hipótesis:

H1 $H : \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \equiv_{\mathsf{R}} H^{@} : e_0^{@}$
H2 $H : \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \Downarrow_{A,C} K : w$
H3 Por la regla *Case*:
a) $H : p \Downarrow_{A,C} L : C_k\ \overline{p_j}$
b) $L : e_k[\overline{p_j/x_{kj}}] \Downarrow_{A,C} K : w$
H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0^{@} = \mathbf{case}\ q\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}}$
P2 Por H1 y P1:
$H : p \equiv_{\mathsf{R}} H^{@} : q$
P3 Por H1, P2, H4 y h.i. sobre H3(a):
a) $H^{@} : q \Downarrow_{A,C} L^{@} : w_0^{@}$
b) $L : C_k\ \overline{p_j} \equiv_{\mathsf{R}} L^{@} : w_0^{@}$
c$_1$) $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$
c$_2$) $L : \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}} \equiv_{\mathsf{R}} L^{@} : e_0^{@}$
P4 Por P3(b):
$w_0^{@} = C_k\ \overline{q_j}$
P5 Por P3(b), P4 y P3(c$_2$):
$L : e_k[\overline{p_j/x_{kj}}] \equiv_{\mathsf{R}} L^{@} : e_k^{@}[\overline{q_j/x_{kj}}]$
P6 Por P5, P3(c$_1$) y h.i. sobre H3(b):
a) $L^{@} : e_k^{@}[\overline{q_j/x_{kj}}] \Downarrow_{A,C} K^{@} : w^{@}$
b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)
c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)
P7 Por P3(a), P4, P6(a) y la regla *Case*:
$H^{@} : \mathbf{case}\ q\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}} \Downarrow_{A,C} K^{@} : w^{@}$ (T1)

App

Hipótesis:

H1 $H : p\ q \equiv_{\mathsf{R}} H^{@} : e_0^{@}$
H2 $H : p\ q \Downarrow_{A,C} K : w$
H3 Por la regla *App*:

a) $H : p \Downarrow_{A,C} L : \lambda x.e$

b) $L : e[q/x] \Downarrow_{A,C} K : w$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0^{@} = p'\ q'$

P2 Por H1 y P1:

a) $H : p \equiv_{\mathsf{R}} H^{@} : p'$

b) $H : q \equiv_{\mathsf{R}} H^{@} : q'$

P3 Por P2, H4 y h.i. sobre H3(a):

a) $H^{@} : p' \Downarrow_{A,C} L^{@} : w_0^{@}$

b) $L : \lambda x.e \equiv_{\mathsf{R}} L^{@} : w_0^{@}$

c_1) $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$

c_2) $L : q \equiv_{\mathsf{R}} L^{@} : q'$

P4 Por P3(b):
$w_0^{@} = \lambda^{@} x.e^{@}$

P5 Por P3(b), P4, P3(c_2) y la Propiedad 6.1.6:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} : e^{@}[q'/x]$

P6 Por P5, P3(c_1) y h.i. sobre H3(b):

a) $L^{@} : e^{@}[q'/x] \Downarrow_{A,C} K^{@} : w^{@}$

b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2, case $w_0^{@} = \lambda x.e^{@}$)

c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3, case $w_0^{@} = \lambda x.e^{@}$)

Ahora por casos sobre P4:

$w_0^{@} = \lambda x.e^{@}$

P7 Por P3(a), P4, P6(a) y la regla *App*:
$H^{@} :\ p'\ q' \Downarrow_{A,C} K^{@} : w^{@}$ (T1)

$w_0^{@} = \lambda^{@(r,s)} x.e^{@}$

P7 Propiedad 6.1.4 aplicada a P5:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} \cup [q'' \mapsto q'^{@(l,0)}] : (e^{@}[q'/x])[q''/q']$

P8 Propiedad 6.1.5 aplicada a P7:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} \cup [q''' \mapsto e^{@}[q''/x],\ q'' \mapsto q'^{@(l,0)}] : q'''^{@(l,1)}$

P9 Por P3(c_1), P8, Definición 6.3 y q'', q''' son frescas:
$L : e' \equiv_{\mathsf{R}} L^{@} \cup [q''' \mapsto e^{@}[q''/x],\ q'' \mapsto q'^{@(l,0)}] : e'^{@}$

P10 Por P8, P9 y h.i. sobre H3(b):

a) $L^{@} \cup [q''' \mapsto e^{@}[q''/x], q'' \mapsto q'^{@(l,0)}] : q'''^{@(l,1)} \Downarrow_{A,C} K'^{@} : w'^{@}$

b) $K : w \equiv_{\mathsf{R}} K'^{@} : w'^{@}$ (T2)

c) $K : e' \equiv_{\mathsf{R}} K'^{@} : e'^{@}$ (T3)

P11 Aplicando la regla *App*@ a P3(a) y P10(a):
$H^{@} :\ p'\ q' \Downarrow_{A,C} K'^{@} : w'^{@}$ (T1)

Var′

Hipótesis:

H1 $H : p \equiv_{\mathsf{R}} \ H^{@} : e_0^{@}$

H2 $H : p \ \Downarrow_{A,C} \ K \diamond [p \mapsto w] : w$

H3 Por la regla *Var′*:

a) $(p \mapsto e) \in H$

b) $H : e \ \Downarrow_{A,C} \ K : w$

H4 $H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@}$

Demostración:

P1 Por H1:

a) $\exists n \geq 0 \ | H^{@} : e_0^{@} = H_0^{@} \cup [\overline{q_i \mapsto q_{i-1}^{@}}^{n}] : q_n$

b) $q_0 = e^{@}$

c) $rp \ (e) = \mathsf{R} \ (rp \ (e^{@}))$

P2 Por H1 y P1:
$H : e \equiv_{\mathsf{R}} \ H^{@} : e^{@}$

P3 Por P2, H4 y h.i. sobre H3(b):

a) $H^{@} : e^{@} \ \Downarrow_{A,C} \ K_0^{@} : w^{@}$

b) $K : w \equiv_{\mathsf{R}} \ K_0^{@} : w^{@}$

c) $K : e' \equiv_{\mathsf{R}} \ K_0^{@} : e'^{@}$

P4 Propiedad 6.1.7 aplicada a P3(a) y P1:

a) $H^{@} : q_n \ \Downarrow_{A,C} \ K_0^{@} \diamond [\overline{q_i \mapsto w^{@i}}^{n}, \overline{q'_i \mapsto q''^{@}_i}] : w^{@0}$ (T1)

b) $\forall i \ \exists j \ | w = (\mathsf{R} \ K)^j w^i$

c) $\overline{q'_i}$ son frescos

P5 Por H1, H3(a), P1, P3(b) y P4(b):
$K \diamond [p \mapsto w] : w \equiv_{\mathsf{R}} \ K_0^{@} \diamond [\overline{q_i \mapsto w^{@i}}^{n}, \ \ldots] : w^{@0}$ (T2)

P6 Por P4(b,c), P3(c) y P1:
$K \diamond [p \mapsto w] : e' \equiv_{\mathsf{R}} \ K_0^{@} \diamond [\overline{q_i \mapsto w'^{@i}}^{n}, \ \ldots] : e'^{@}$ (T3)

OpP

Hipótesis:

H1 $H : op \ p_1 \ p_2 \equiv_{\mathsf{R}} \ H^{@} : e_0^{@}$

H2 $H : op \ p_1 \ p_2 \ \Downarrow_{A,C} \ K : m_1 \ op \ m_2$

H3 Por la regla *OpP*:

a) $H : p_1 \ \Downarrow_{A,C} \ L : m_1$

b) $L : p_2 \ \Downarrow_{A,C} \ K : m_2$

H4 $H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0^@ = op\ q_1\ q_2$
P2 Por H1 y P1:
$H : p_i \equiv_\mathsf{R}\ H^@ : q_i$
P3 Por P2, H4 y h.i. sobre H3(a):
a) $H^@ : p_1\ \Downarrow_{A,C}\ L^@ : w_1^@$
b) $L : m_1 \equiv_\mathsf{R}\ L^@ : w_1^@$
c$_1$) $L : e' \equiv_\mathsf{R}\ L^@ : e'^@$
c$_2$) $L : op\ p_1\ p_2 \equiv_\mathsf{R}\ L^@ : e_0^@$
P4 Por P3(b):
$w_1^@ = m_1$
P5 Por P3(c$_2$):
$L : p_2 \equiv_\mathsf{R}\ L^@ : q_2$
P6 Por P5, P3(c$_1$) y h.i. sobre H3(b):
a) $L^@ : q_2\ \Downarrow_{A,C}\ K^@ : w_2^@$
b) $K : m_2 \equiv_\mathsf{R}\ K^@ : w_2^@$
c) $K : e' \equiv_\mathsf{R}\ K^@ : e'^@$ (T3)
P7 Por P6(b):
$w_2^@ = m_2$
P8 Por P3(a), P4, P6(a), P7 y la regla *OpP*:
$H^@ : op\ q_1\ q_2\ \Downarrow_{A,C}\ K^@ : m_1\ op\ m_2$ (T1)
P9 Por la Definición 6.3:
$K : m_1\ op\ m_2 \equiv_\mathsf{R}\ K^@ : m_1\ op\ m_2$ (T2)

Demostración de la segunda implicación: $\Leftarrow$

Lam **,** *Lam@* **o** *Cons*

Hipótesis:
H1 $H : e \equiv_\mathsf{R}\ H^@ : e^@$
H2 $H^@ : e^@\ \Downarrow_{A,C}\ K^@ : w^@$
H3 Por la regla *Lam*, *Lam@* o *Cons*:
a) $K^@ : w^@ = H^@ : e^@$
b) $e^@ = \lambda^@ x.e'^@$ o $e = C\ \overline{q_i}$
H4 $H : e' \equiv_\mathsf{R}\ H^@ : e'^@$
Demostración:
P1 Por H1 y H3(b):
$e = \lambda x.e'$ o $e = C\ \overline{p_i}$
P2 Por P1 y la regla *Lam* o *Cons*:
$H : e\ \Downarrow_{A,C}\ H : e$ (T1)

P3 Por H1 y H3(a):
$H : e \equiv_{\mathsf{R}} \ H^{@} : e^{@} = K^{@} : w^{@}$ (T2)

P4 Por H4 y H3(a):
$H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@} = K^{@} : e'$ (T3)

Letrec

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} \ H^{@} : \mathbf{letrec}\ \overline{x_i = e_i^{@}}\ \mathbf{in}\ e^{@}$

H2 $H^{@} : \mathbf{letrec}\ \overline{x_i = e_i^{@}}\ \mathbf{in}\ e^{@} \ \Downarrow_{A,C} \ K^{@} : w^{@}$

H3 Por la regla *Letrec*:

a) $H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}} \ \Downarrow_{A,C} \ K^{@} : w^{@}$

b) $\overline{q_i}$ frescos con respecto a $H^{@}$, C, A y **letrec** $\overline{x_i = e_i^{@}}$ **in** $e^{@}$

H4 $H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0 = \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e$

P2 Por H1, H3(b), H4, P1 y Definición 6.3:

a) $H \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \equiv_{\mathsf{R}} \ H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : \hat{e^{@}}$ si $\overline{p_i}$ son frescas

c) $H \cup [\overline{p_i \mapsto \hat{e_i}}] : e' \equiv_{\mathsf{R}} \ H^{@} \cup [\overline{q_i \mapsto \hat{e_i^{@}}}] : e'^{@}$ si $\overline{p_i}$ son frescas

P3 Por P2 y h.i. sobre H3(a):

a) $H \cup [\overline{p_i \mapsto \hat{e_i}}] : \hat{e} \ \Downarrow_{A,C} \ K : w$

b) $K : w \equiv_{\mathsf{R}} \ K^{@} : w^{@}$ (T2)

c) $K : e' \equiv_{\mathsf{R}} \ K^{@} : e'^{@}$ (T3)

P4 Por P3(a) y la regla *Letrec*:
$H : \ \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \ \Downarrow_{A,C} \ K : w$ (T1)

Case

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} \ H^{@} : \mathbf{case}\ q\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}}$

H2 $H^{@} : \mathbf{case}\ q\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}} \ \Downarrow_{A,C} \ K^{@} : w^{@}$

H3 Por la regla *Case*:

a) $H^{@} : q \ \Downarrow_{A,C} \ L^{@} : C_k\ \overline{q_j}$

b) $L^{@} : e_k^{@}[\overline{q_j/x_{kj}}] \ \Downarrow_{A,C} \ K^{@} : w^{@}$

H4 $H : e' \equiv_{\mathsf{R}} \ H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0 = \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}$

P2 Por H1 y P1:
$H : p \equiv_{\mathsf{R}} H^{@} : q$

P3 Por H1, P2, H4 y h.i. sobre H3(a):

a) $H : p \Downarrow_{A,C} L : w_0$

b) $L : w_0 \equiv_{\mathsf{R}} L^{@} : C_k\ \overline{q_j}$

c$_1$) $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$

c$_2$) $L : e_0 \equiv_{\mathsf{R}} L^{@} : \mathbf{case}\ q\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i^{@}}$

P4 Por P3(b):
$w_0 = C_k\ \overline{p_j}$

P5 Por P3(b), P4 y P3(c$_2$):
$L : e_k[\overline{p_j/x_{kj}}] \equiv_{\mathsf{R}} L^{@} : e_k^{@}[\overline{q_j/x_{kj}}]$

P6 Por P5, P3(c$_1$) y h.i. sobre H3(b):

a) $L : e_k[\overline{p_j/x_{kj}}] \Downarrow_{A,C} K : w$

b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)

c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P7 Por P3(a), P4, P6(a) y la regla *Case*:
$H :\ \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \Downarrow_{A,C} K : w$ (T1)

App

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : p'\ q'$

H2 $H^{@} : p'\ q' \Downarrow_{A,C} K^{@} : w^{@}$

H3 Por la regla *App*:

a) $H^{@} : p' \Downarrow_{A,C} L^{@} : \lambda x.e^{@}$

b) $L^{@} : e^{@}[q'/x] \Downarrow_{A,C} K^{@} : w^{@}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0 = p\ q$

P2 Por H1 y P1:
$H : p \equiv_{\mathsf{R}} H^{@} : p'$

P3 Por P2, H4 y h.i. sobre H3(a):

a) $H : p \Downarrow_{A,C} L : w_0$

b) $L : w_0 \equiv_{\mathsf{R}} L^{@} : \lambda x.e^{@}$

c$_1$) $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$

c$_2$) $L : q \equiv_{\mathsf{R}} L^{@} : q'$

P4 Por P3(b):
$w_0 = \lambda x.e$

P5 Por P3(b), P4, P3(c$_2$) y Propiedad 6.1.6:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} : e^{@}[q'/x]$

P6 Por P5, P3(c_1) y h.i. sobre H3(b):
- **a)** $L : e[q/x] \Downarrow_{A,C} K : w$
- **b)** $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)
- **c)** $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P7 Por P3(a), P4, P6(a) y la regla *Var*:
$H : p\ q \Downarrow_{A,C} K : w$ (T1)

App@

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : p'\ q'$

H2 $H^{@} : p'\ q' \Downarrow_{A,C} K^{@} : w^{@}$

H3 Por la regla *App@*:
- **a)** $H^{@} : p' \Downarrow_{A,C} L^{@} : \lambda^{@(r,s)}x.e^{@}$
- **b)** $L^{@} \cup [q''' \mapsto e^{@}[q''/x],\ q'' \mapsto q'^{@(l,0)}] : q'''^{@(l,1)} \Downarrow_{A,C} K^{@} : w^{@}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0 = p\ q$

P2 Por H1 y P1:
$H : p \equiv_{\mathsf{R}} H^{@} : p'$

P3 Por P2, H4 y h.i. sobre H3(a):
- **a)** $H : p \Downarrow_{A,C} L : w_0$
- **b)** $L : w_0 \equiv_{\mathsf{R}} L^{@} : \lambda^{@(r,s)}x.e^{@}$
- **c_1)** $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$
- **c_2)** $L : q \equiv_{\mathsf{R}} L^{@} : q'$

P4 Por P3(b):
$w_0 = \lambda x.e$

P5 Por P3(b), P4 y P3(c_2):
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} : e^{@}[q'/x]$

P6 Propiedad 6.1.4 aplicado a P5:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} \cup [q'' \mapsto q^{@(l,0)}] : (e^{@}[q'/x])[q''/q']$

P7 Propiedad 6.1.5 aplicado a P6:
$L : e[q/x] \equiv_{\mathsf{R}} L^{@} \cup [q''' \mapsto e^{@}[q''/x],\ q'' \mapsto q'^{@(l,0)}] : q'''^{@(l,1)}$

P8 Por P7 y h.i. sobre H3(b):
- **a)** $L : e[q/x] \Downarrow_{A,C} K : w$
- **b)** $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)
- **c)** $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P9 Por P3(a), P4, P8(a) y la regla *App*:
$H : e_0 \Downarrow_{A,C} K : w$ (T1)

Var′

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : q$

H2 $H^{@} : q \Downarrow_{A,C} K^{@} \diamond [q \mapsto w^{@}] : w^{@}$

H3 Por la regla *Var′*:

a) $(q \mapsto e^{@}) \in H^{@}$

b) $H^{@} : e^{@} \Downarrow_{A,C} K^{@} : w^{@}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1 y H3(a):
$H : e_0 = H[p \mapsto e] : p$
Ahora procedemos analizando por casos sobre $e^{@}$ y e. Nótese que es imposible que $e^{@} \neq q'^{@}$ y $e = p'$, por definición de $\equiv_{\mathsf{R}}$:

$e^{@} = q'^{@}$ **y** $e \neq p'$

P2 Por H1 y P1:
$H : e_0 \equiv_{\mathsf{R}} H^{@} : q'^{@}$

P3 Por P2, H4 y h.i. sobre H3(b):

a) $H : e_0 \Downarrow_{A,C} K : w$ (T1)

b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$

c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$

P4 Por P3(b,c), Proposición 6.3, H1 y P1:

a) $K : w \equiv_{\mathsf{R}} K^{@} \diamond [q \mapsto w^{@}] : w^{@}$ (T2)

b) $K : e' \equiv_{\mathsf{R}} K^{@} \diamond [q \mapsto w^{@}] : e'^{@}$ (T3)

$(e^{@} \neq q'^{@}$ **y** $e \neq p')$ **o** $(e^{@} = q'^{@}$ **y** $e = p')$

P2 Por P1, H1 y H3(a):
$H : e \equiv_{\mathsf{R}} H^{@} : e^{@}$

P3 Por P2, H4 y h.i. sobre H3(b):

a) $H : e \Downarrow_{A,C} K : w$

b) $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$

c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$

P4 Por P3(a) y aplicando la regla *Var′*:
$H[p \mapsto e] : p \Downarrow_{A,C} K \diamond [p \mapsto w] : w$ (T1)

P5 Por P3(b), H1 y P1:

a) $K \diamond [p \mapsto w] : w \equiv_{\mathsf{R}} K^{@} \diamond [q \mapsto w^{@}] : w^{@}$ (T2)

b) $K \diamond [p \mapsto w] : e' \equiv_{\mathsf{R}} K^{@} \diamond [q \mapsto w^{@}] : e'^{@}$ (T3)

OpP

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : op\ q_1\ q_2$

H2 $H^{@} : op\ q_1\ q_2 \Downarrow_{A,C} K^{@} : m_1\ op\ m_2$

H3 Por la regla *OpP*:

a) $H^{@} : q_1 \Downarrow_{A,C} L^{@} : m_1$

b) $L^{@} : q_2 \Downarrow_{A,C} K^{@} : m_2$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$e_0 = op\ p_1\ p_2$

P2 Por H1 y P1:
$H : p_i \equiv_{\mathsf{R}} H^{@} : q_i$

P3 Por P2, H4 y h.i. sobre H3(a):

a) $H : p_1 \Downarrow_{A,C} L : w_1$

b) $L : w_1 \equiv_{\mathsf{R}} L^{@} : m_1$

c$_1$) $L : e' \equiv_{\mathsf{R}} L^{@} : e'^{@}$

c$_2$) $L : e_0 \equiv_{\mathsf{R}} L^{@} : op\ q_1\ q_2$

P4 Por P3(b):
$w_1 = m_1$

P5 Por P3(c$_2$):
$L : p_2 \equiv_{\mathsf{R}} L^{@} : q_2$

P6 Por P5, P3(c$_1$) y h.i. sobre H3(b):

a) $L : p_2 \Downarrow_{A,C} K : w_2$

b) $K : w_2 \equiv_{\mathsf{R}} K^{@} : m_2$

c) $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P7 Por P6(b):
$w_2 = m_2$

P8 Por P3(a), P4, P6(a), P7 y la regla *OpP*:
$H : op\ p_1\ p_2 \Downarrow_{A,C} K : m_1\ op\ m_2$ (T1)

P9 Por la Definición 6.3:
$K : m_1\ op\ m_2 \equiv_{\mathsf{R}} K^{@} : m_1\ op\ m_2$ (T2)

Var@S

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : q^{@str}$

H2 $H^{@} : q^{@str} \Downarrow_{A,C} K^{@} : w^{@}$

H3 Por la regla *Var@S*:
$H^{@} : q^{@(r,s)} \Downarrow_{A,C} K^{@} : w^{@}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$H : e_0 \equiv_{\mathsf{R}} H^{@} : q^{@(r,s)}$

P2 Por P1, H4 y h.i. sobre H3:
- **a)** $H : e_0 \Downarrow_{A,C} K : w$ (T1)
- **b)** $K : w \equiv_{\mathsf{R}} K^{@} : w^{@}$ (T2)
- **c)** $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

Var@C

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : q^{@(r,s)}$

H2 $H^{@} : q^{@(r,s)} \Downarrow_{A,C} K^{@} \cup \overline{[q'_i \mapsto q_i^{@(l,i)}]} : C\ \overline{q'_i}$

H3 Por la regla *Var@C*:
- **a)** $H^{@} : q \Downarrow_{A,C} K^{@} : C\ \overline{q_i}$
- **b)** $\overline{q'_i}$ fresco con respecto a $K^{@}$, A, C y $q^{@(r,s)}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$H : e_0 \equiv_{\mathsf{R}} H^{@} : q$

P2 Por P1, H4 y h.i. sobre H3(a):
- **a)** $H : e_0 \Downarrow_{A,C} K : w$ (T1)
- **b)** $K : w \equiv_{\mathsf{R}} K^{@} : C\ \overline{q_i}$
- **c)** $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$

P3 Por P2(b), H3(b) y la Propiedad 6.1.5:
- **a)** $K : w \equiv_{\mathsf{R}} K^{@} \cup \overline{[q'_i \mapsto q_i^{@(l,i)}]} : C\ \overline{q'_i}$ (T2)
- **b)** $K : e' \equiv_{\mathsf{R}} K^{@} \cup \overline{[q'_i \mapsto q_i^{@(l,i)}]} : e'^{@}$(T3)

Var@F **y** *Var@FO*

Hipótesis:

H1 $H : e_0 \equiv_{\mathsf{R}} H^{@} : q^{@(r,s)}$

H2 $H^{@} : q^{@(r,s)} \Downarrow_{A,C} K^{@} : \lambda^{@(r,s)} x.e^{@}$

H3 Por la regla *Var@F*:
$H^{@} : q \Downarrow_{A,C} K^{@} : \lambda x.e^{@}$

H4 $H : e' \equiv_{\mathsf{R}} H^{@} : e'^{@}$

Demostración:

P1 Por H1:
$H : e_0 \equiv_{\mathsf{R}} H^{@} : q$

P2 Por P1, H4 y h.i. sobre H3:
- **a)** $H : e_0 \Downarrow_{A,C} K : w$ (T1)
- **b)** $K : w \equiv_{\mathsf{R}} K^{@} : \lambda^{@} x.e^{@}$
- **c)** $K : e' \equiv_{\mathsf{R}} K^{@} : e'^{@}$ (T3)

P3 Por P2(b):
$K : w \equiv_{\mathsf{R}} K^{@} : \lambda^{@(r,s):\cdots} x.e^{@}$ (T2)

□

Demostración 13 (Proposición 6.7) Demostración de la primera implicación: ⇒
Esta demostración se realiza por inducción sobre las reglas. Es muy sencilla, ya que sólo es necesario aplicar hipótesis de inducción y realizar una composición de los resultados.

Lam, *Lam@* **o** *Cons*

Hipótesis:

H1 $H : e \looparrowright f \ \Downarrow_{A,C} \ H : e \looparrowright f$

Demostración:

$(H, e, S, f) \Longrightarrow^{0} (H, e, S, f)$

Letrec

Hipótesis:

H1 $H : \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e \looparrowright f \ \Downarrow_{A,C} \ K : w \looparrowright f'$

H2 Por la regla *Letrec*:

a) $H \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \looparrowright f \ \Downarrow_{A,C} \ K : w \looparrowright f'$

b) $\overline{p_i}$ fresco con respecto a H, A, C y $\mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e$

Demostración:

	$(H, \mathbf{letrec}\ \overline{x_i = e_i}\ \mathbf{in}\ e, S, f)$	$\{letrec\}$
$\Longrightarrow$	$(H \cup [\overline{p_i \mapsto \hat{e}_i}], \hat{e}, S, f')$	{h.i. sobre H2(a)}
$\Longrightarrow^{*}$	(K, w, S, f')	$\surd$

Case

Hipótesis:

H1 $H : \mathbf{case}\ p\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i} \looparrowright f \ \Downarrow_{A,C} \ K : w \looparrowright f''$

H2 Por la regla *Case*:

a) $H : p \looparrowright f \ \Downarrow_{A,C} \ L : C_k\ \overline{p_j} \looparrowright f'$

b) $L : e_k[\overline{p_j / x_{kj}}] \looparrowright f' \ \Downarrow_{A,C} \ K : w \looparrowright f''$

Demostración:

	$(H, \mathbf{case}\ p\ \mathbf{of}\ \overline{alt_i}, S, f)$	$\{case1\}$
$\Longrightarrow$	$(H, p, \overline{alt_i} : S, f')$	{h.i. sobre H2(a)}
$\Longrightarrow^{*}$	$(L, C_k\ \overline{p_j}, \overline{alt_i} : S, f')$	$\{case2\}$
$\Longrightarrow$	$(L, e_k[\overline{p_j / x_{kj}}], S, f'')$	{h.i. sobre H2(b)}
$\Longrightarrow^{*}$	(K, w, S, f'')	$\surd$

App

Hipótesis:

H1 $H : p\ q \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f''$

H2 Por la regla *App*:

a) $H : p \rightarrowtail f \ \Downarrow_{A,C} \ L : \lambda x.e \rightarrowtail f'$

b) $L : e[p/x] \rightarrowtail f' \ \Downarrow_{A,C} \ K : w \rightarrowtail f''$

Demostración:

	$(H, p\ q, S)$	$\{app1\}$
$\Longrightarrow$	$(H, p, q : S, f)$	{h.i. sobre H2(a)}
$\Longrightarrow^*$	$(L, \lambda x.e, q : S, f')$	$\{app2\}$
$\Longrightarrow$	$(L, e[q/x], S, f')$	{h.i. sobre H2(b)}
$\Longrightarrow^*$	(K, w, S, f'')	$\surd$

Var

Hipótesis:

H1 $H \cup [p \mapsto e] : p \rightarrowtail f \ \Downarrow_{A,C} \ K \cup [p \mapsto w] : w \rightarrowtail f'$

H2 Por la regla *Var*:

$H : e \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f'$

Demostración:

	$(H \cup [p \mapsto e], p, S)$	$\{var1\}$
$\Longrightarrow$	$(H, e, \#p : S, f)$	{h.i. sobre H2}
$\Longrightarrow^*$	$(K, w, \#p : S, f')$	$\{var2$o $var3\}$
$\Longrightarrow^*$	$(K \cup [p \mapsto w], w, S, f')$	$\surd$

OpP

Hipótesis:

H1 $H : op\ p_1\ p_2 \rightarrowtail f \ \Downarrow_{A,C} \ K : m_1\ op\ m_2 \rightarrowtail f'$

H2 Por la regla *OpP*:

a) $H : p_1 \rightarrowtail f \ \Downarrow_{A,C} \ L : m_1 \rightarrowtail f'$

b) $L : e \rightarrowtail f' \ \Downarrow_{A,C} \ K : m_2 \rightarrowtail f'$

Demostración:

	$(H, op\ p_1\ p_2, S)$	$\{op\#\}$
$\Longrightarrow$	$(H, \textbf{case}\ p_1\ \textbf{of}\ Int\ p_1 \mapsto \textbf{case}\ p_2\ \textbf{of}\ Int\ p_2 \mapsto p_1\ op\ p_2, S, f)$	$\{case1\}$
$\Longrightarrow$	$(H, p_1, alts : S, f)$	{h.i. sobre H2(a)}
$\Longrightarrow^*$	$(H, m_1, alts : S, f)$	$\{case2\}$
$\Longrightarrow$	$(H, \textbf{case}\ p_2\ \textbf{of}\ Int\ p_2 \mapsto m_1\ op\ p_2, S, f)$	$\{case1\}$
$\Longrightarrow$	$(H, p_2, alts' : S, f)$	{h.i. sobre H2(b)}
$\Longrightarrow^*$	$(H, m_2, alts' : S, f)$	$\{case2\}$
$\Longrightarrow^*$	$(H, m_1\ op\ m_2, S, f)$	$\surd$

Var@S

Hipótesis:

H1 $H : q^{@str} \looparrowright f \ \Downarrow_{A,C} \ K : w \looparrowright f'$

H2 Por la regla *Var@S*:
$H : q^{@(r,s)} \looparrowright f \circ \langle 0\,0\ Observe\ str \rangle \ \Downarrow_{A,C} \ K : w \looparrowright f'$

Demostración:

	$(H,\ q^{@str}, S, f)$	$\{var@S\}$
$\Longrightarrow$	$(H, q^{@(n,0)}, S, f \circ \langle 0\,0\ Observe\ str \rangle)$	{h.i. sobre H2}
$\Longrightarrow^*$	(K, w, S, f')	$\surd$

Var@C

Hipótesis:

H1 $H : \ p^{@(r,s)} \looparrowright f \ \Downarrow_{A,C} \ K \cup [\overline{q'_i \mapsto q_i^{@(length\ f',i)}}] : C\ \overline{q'_i} \looparrowright f' \circ \langle r\ s\ Cons\ k\ C \rangle$

H2 Por la regla *Var@C*:
$H : p \looparrowright f \circ \langle r\ s\ Enter \rangle \ \Downarrow_{A,C} \ K : C\ \overline{q_i} \looparrowright f'$

Demostración:

	$(H,\ p^{@(r,s)}, S, f)$	$\{var1@\}$
$\Longrightarrow$	$(H, p, @(r,s) : S, f \circ \langle r\ s\ Enter \rangle)$	{h.i. sobre H2}
$\Longrightarrow^*$	$(K, C\ \overline{q_i}, @(r,s) : S, f')$	$\{var3@\}$
$\Longrightarrow^*$	$(K \cup [q'_i \mapsto \overline{q_i^{@(length\ f',i)}}], C\ \overline{q'_i}, S, f' \circ \langle r\ s\ Cons\ k\ C \rangle)$	$\surd$

Var@F

Hipótesis:

H1 $H : q^{@(r,s)} \looparrowright f \ \Downarrow_{A,C} \ K : \lambda^{@(r,s)} x.e \looparrowright f'$

H2 Por la regla *Var@F*:
$H : q \looparrowright f \circ \langle r\ s\ Enter \rangle \ \Downarrow_{A,C} \ K : \lambda x.e \looparrowright f'$

Demostración:

	$(H,\ q^{@(r,s)}, S, f)$	$\{var1@\}$
$\Longrightarrow$	$(H, q, @(r,s) : S, f \circ \langle r\ s\ Enter \rangle)$	{h.i. sobre H2}
$\Longrightarrow^*$	$(K, \lambda x.e, @(r,s) : S, f')$	$\{var2@\}$
$\Longrightarrow^*$	$(K, \lambda^{@(r,s)} x.e, S, f')$	$\surd$

Var@FO

Hipótesis:

H1 $H : q^{@(r,s)} \looparrowright f \ \Downarrow_{A,C} \ K : \lambda^{@(r,s):obs} x.e \looparrowright f'$

H2 Por la regla *Var@F*:
$H : q \looparrowright f \circ \langle r\ s\ Enter \rangle \ \Downarrow_{A,C} \ K : \lambda^{@obs} x.e \looparrowright f'$

Demostración:

	$(H,\ q^{@(r,s)}, S, f)$	$\{var1@\}$
$\Longrightarrow$	$(H, q, @(r,s) : S, f \circ \langle r\ s\ Enter \rangle)$	{h.i. sobre H2}
$\Longrightarrow^*$	$(K, \lambda^{@obs} x.e, @(r,s) : S, f')$	$\{var2@\}$
$\Longrightarrow^*$	$(K, \lambda^{@(r,s):obs} x.e, S, f')$	$\surd$

App@

Hipótesis:

H1 $H : e\ p \rightarrowtail f\ \Downarrow_{A,C}\ K : w \rightarrowtail f''$

H2 Por la regla *App@*:

a) $H : e \rightarrowtail f\ \Downarrow_{A,C}\ L : \lambda^{@(r,s)}x.e' \rightarrowtail f'$

b) $L \cup [q' \mapsto e'[q''/x],\ q'' \mapsto p^{@(length\ f',0)}] : q'^{@(length\ f',1)} \rightarrowtail f' \circ \langle r\ s\ Fun \rangle\ \Downarrow_{A,C}\ K : w \rightarrowtail f''$

Demostración:

	$(H, e\ p, S, f)$	$\{app1\}$
$\Longrightarrow$	$(H, e, p : S, f)$	{h.i. sobre H2(a)}
$\Longrightarrow^*$	$(L, \lambda^{@(r,s)}x.e', p : S, f')$	$\{app2@\}$
$\Longrightarrow$	$(L \cup \left[\begin{array}{l} q \mapsto e'[q_1/y], \\ q_1 \mapsto p^{@(n',0)}] \end{array}\right], q^{@(n',1)}, S, f' \circ \langle r\ s\ Fun \rangle)$	{h.i. sobre H2(b)}
$\Longrightarrow^*$	(K, w, S, f'')	$\surd$

Demostración de la segunda implicación: $\Leftarrow$

Como los cómputos balanceados se corresponden con trazas balanceadas, se puede realizar la demostración por casos sobre las trazas balanceadas.

ϵ

Hipótesis:

H1 $(H, e, S, f) \Longrightarrow^0 (K, w, S, f')$

Demostración:

P1 Por H1:

a) $(H, e, S, f) = (K, w, S, f')$

b) $w = C\ \overline{p_i}$ o $w = \lambda^{@}x.e'$

P2 Por P1 y aplicando la regla *Cons*, *App* o *App@*:
$H : e \rightarrowtail f\ \Downarrow_{A,C}\ H : e \rightarrowtail f\ \surd$

app1 bal app2 bal

Hipótesis:

	$(H, p\ q, S, f)$	$\{app1\}$
$\Longrightarrow$	$(H, p, q : S, f)$	$\{\text{bal}_1\}$
$\Longrightarrow^*$	$(L, \lambda x.e, q : S, f')$	$\{app2\}$
$\Longrightarrow$	$(L, e[q/x], S, f')$	$\{\text{bal}_2\}$
$\Longrightarrow^*$	(K, w, S, f'')	

Demostración:

P1 Por h.i. sobre bal_1:
$H : p \rightarrowtail f \;\Downarrow_{A,C}\; L : \lambda x.e \rightarrowtail f'$

P2 Por h.i. sobre bal_2: $L : e[q/x] \rightarrowtail f' \;\Downarrow_{A,C}\; K : w \rightarrowtail f''$

P3 Por P1, P2 y la regla *App*:
$H : e \rightarrowtail f \;\Downarrow_{A,C}\; K : w \rightarrowtail f'' \;\surd$

var1 bal var2 **o** *var1 bal var3*

Hipótesis:

$$\begin{array}{lll} & (H \cup [p \mapsto e], p, S, f) & \{var1\} \\ \Longrightarrow & (H, e, \#p : S, f) & \{\text{bal}\} \\ \Longrightarrow^* & (K, w, \#p : S, f') & \{var2 \text{o } var3\} \\ \Longrightarrow^* & (K \cup [p \mapsto w], w, S, f') & \end{array}$$

Demostración:

P1 Por h.i. sobre *bal*:
$H : e \rightarrowtail f \;\Downarrow_{A,C}\; K : w \rightarrowtail f'$

P2 Por P1 y la regla *Var*:
$H \cup [p \mapsto e] : p \rightarrowtail f \;\Downarrow_{A,C}\; K \cup [p \mapsto w] : w \rightarrowtail f' \;\surd$

letrec bal

Hipótesis:

$$\begin{array}{lll} & (H, \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e, S, f) & \{letrec\} \\ \Longrightarrow & (H \cup [\overline{p_i \mapsto \hat{e}_i}], \hat{e}, S, f) & \{\text{bal}\} \\ \Longrightarrow^* & (K, w, S, f') & \end{array}$$

Demostración:

P1 Por h.i. sobre *bal*:

a) $H \cup [\overline{p_i \mapsto \hat{e}_i}] : \hat{e} \rightarrowtail f \;\Downarrow_{A,C}\; K : w \rightarrowtail f'$

b) $\overline{p_i}$ fresco con respecto a H

P2 Por P1 y la regla *Letrec*:
$H : \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e \rightarrowtail f \;\Downarrow_{A,C}\; K : w \rightarrowtail f' \;\surd$

case1 bal_1 *case2* bal_2

Hipótesis:

$$\begin{array}{lll} & (H, \textbf{case } p \textbf{ of } \overline{alt_i}, S, f) & \{case1\} \\ \Longrightarrow & (H, p, \overline{alt_i} : S, f) & \{\text{bal}_1\} \\ \Longrightarrow^* & (L, C_k\ \overline{p_j}, \overline{alt_i} : S, f') & \{case2\} \\ \Longrightarrow & (L, e_k[\overline{p_j/x_{kj}}], S, f') & \{\text{bal}_2\} \\ \Longrightarrow^* & (K, w, S, f'') & \end{array}$$

Demostración:

P1 Por h.i. sobre bal_1:
$H : p \rightarrowtail f \;\Downarrow_{A,C}\; L : C_k\ \overline{p_j} \rightarrowtail f'$

P2 Por h.i. sobre bal_2:
$L : e_k[\overline{p_j/x_{kj}}] \rightarrowtail f' \ \Downarrow_{A,C} \ K : w \rightarrowtail f''$
P3 Por P1, P2 y la regla *Case*:
$H :$ **case** p **of** $\overline{C_i \ \overline{x_{ij}} \rightarrow e_i} \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f'' \ \surd$

op#

Hipótesis:

$$\begin{array}{lll} & (H, op\ m_1\ m_2, S, f) & \{op\#\} \\ \Longrightarrow & (H, m_1\ op\ m_2, S, f) & \end{array}$$

Demostración:

P1 Por la regla *OpP*:
$H : op\ m_1\ m_2 \rightarrowtail f \ \Downarrow_{A,C} \ H : m_1\ op\ m_2 \rightarrowtail f \ \surd$

app1 bal app2@ bal

Hipótesis:

$$\begin{array}{lll} & (H, e\ p, S, f) & \{app1\} \\ \Longrightarrow & (H, e, p : S, f) & \{\text{bal}_1\} \\ \Longrightarrow^* & (L, \lambda^{@(r,s)} x.e', p : S, f') & \{app2@\} \\ \Longrightarrow & (L \cup \left[\begin{array}{l} q \mapsto e'[q_1/y], \\ q_1 \mapsto p^{@(n',0)}] \end{array}\right], q^{@(n',1)}, S, f' \circ \langle r\ s\ Fun \rangle) & \{\text{bal}_2\} \\ \Longrightarrow^* & (K, w, S, f'') & \end{array}$$

Demostración:

P1 Por h.i. sobre bal_1:
$H : e \rightarrowtail f \ \Downarrow_{A,C} \ L : \lambda^{@(r,s)} x.e' \rightarrowtail f'$
P2 Por h.i. sobre bal_2:
$L \cup [q' \mapsto e'[q''/x],\ q'' \mapsto p^{@(n',0)}] : q'^{@(n',1)} \rightarrowtail f' \circ \langle r\ s\ Fun \rangle \ \Downarrow_{A,C} \ K : w \rightarrowtail f''$
H1 Por P1, P2 y la regla *App@*:
$H : e\ p \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f'' \ \surd$

var@S bal

Hipótesis:

$$\begin{array}{lll} & (H,\ q^{@str}, S, f) & \{\text{var@S}\} \\ \Longrightarrow & (H, q^{@(n,0)}, S, f \circ \langle 0\,0\ Observe\ str \rangle) & \{\text{bal}\} \\ \Longrightarrow^* & (K, w, S, f') & \end{array}$$

Demostración:

P1 Por h.i. sobre *bal*
$H : q^{@(r,s)} \rightarrowtail f \circ \langle 0\,0\ Observe\ str \rangle \ \Downarrow_{A,C} \ K : w \rightarrowtail f'$
P2 Por P1 y la regla *Var@S*:
$H : q^{@str} \rightarrowtail f \ \Downarrow_{A,C} \ K : w \rightarrowtail f' \ \surd$

var1@ bal var2@

Hipótesis:

$$\begin{array}{lll} & (H,\ q^{@(r,s)}, S, f) & \{var1@\} \\ \Longrightarrow & (H, q, @(r,s) : S, f \circ \langle r\ s\ Enter \rangle) & \{bal\} \\ \Longrightarrow^* & (K, \lambda x.e, @(r,s) : S, f') & \{var2@\} \\ \Longrightarrow^* & (K, \lambda^{@(r,s)} x.e, S, f') & \end{array}$$

Demostración:

P1 Por h.i. sobre *bal*:
$H : q \hookrightarrow f \circ \langle r\ s\ Enter \rangle \ \Downarrow_{A,C}\ K : \lambda x.e \hookrightarrow f'$

P2 Por P1 y la regla $Var@F$:
$H : q^{@(r,s)} \hookrightarrow f \ \Downarrow_{A,C}\ K : \lambda^{@(r,s)} x.e \hookrightarrow f'$ $\surd$

var1@ bal var3@

Hipótesis:

$$\begin{array}{lll} & (H,\ p^{@(r,s)}, S, f) & \{var1@\} \\ \Longrightarrow & (H, p, @(r,s) : S, f \circ \langle r\ s\ Enter \rangle) & \{bal\} \\ \Longrightarrow^* & (K, C\ \overline{q_i}, @(r,s) : S, f') & \{var3@\} \\ \Longrightarrow^* & (K \cup [q_i' \mapsto q_i^{@(length\ f', i)}], C\ \overline{q_i'}, S, f' \circ \langle r\ s\ Cons\ k\ C \rangle) & \end{array}$$

Demostración:

P1 Por h.i. sobre *bal*:
$H : p \hookrightarrow f \circ \langle r\ s\ Enter \rangle \ \Downarrow_{A,C}\ K : C\ \overline{q_i} \hookrightarrow f'$

P2 Por P1 y la regla $Var@C$:
$H : p^{@(r,s)} \hookrightarrow f \ \Downarrow_{A,C}\ K \cup [\overline{q_i' \mapsto q_i^{@(length\ f', i)}}] : C\ \overline{q_i'} \hookrightarrow f' \circ \langle r\ s\ Cons\ k\ C \rangle$ $\surd$

□

A.2. Demostración de las proposiciones del Capítulo 8

En esta sección presentaremos las demostraciones de equivalencia entre la semántica de una expresión observada y la equivalente sin observar tanto en GpH como en Eden.

Demostración 14 (Proposición 8.1) La demostración se realizará viendo que las reglas locales con observación generan configuraciones equivalentes a las reglas sin observación y que las reglas globales mantienen los *heaps* equivalentes. En algún caso será necesario realizar varios pasos en las reglas que contienen las observaciones, mientras que sólo será necesario un único paso para el cómputo en el *heap* equivalente sin observaciones.

Esta demostración es similar a la que realizamos para la versión secuencial, debido a que entre otras cosas sólo poseemos un único *heap*. Las principales diferencias son que ahora las ligaduras del *heap* están etiquetadas con una anotación que indica el estado en que se encuentran, que en vez de tener una expresión en el control tenemos una ligadura en el control, que la propiedad de frescura se deduce ahora del *heap* en vez de necesitar los conjuntos A y C, ya que las ligaduras bajo evaluación se etiquetan con el estado *bloqueado* en vez de eliminarse del *heap*, y que ahora los pasos de evaluación son más cortos, es decir, más cercanos a los que realiza la máquina abstracta. Además, la definición de equivalencia ha cambiado. Por tanto, al igual que hicimos en

la demostración de la versión secuencial, en esta demostración no consideraremos el fichero de observaciones, debido a que dicho fichero no dirige el cómputo.

Sea H_0 un *heap* sin observaciones y sea $H^@$ un *heap* cualquiera tales que $H_0 \equiv_R H^@$. Las reglas locales trabajan con ligaduras activas, por tanto tenemos que hacer evolucionar una de las ligaduras activas de H_0. Sea $H_0 = H' + \{p \overset{A}{\mapsto} e\}$, como $H_0 \equiv_R H^@$ entonces $\exists n,\ H^@ = H'^@ + \{\overline{q_i \overset{\alpha_i}{\mapsto} q_{i+1}}^{n-1},\ q_n \overset{\alpha_n}{\mapsto} e^@\}$ donde $rp\ e \equiv_R rp\ e^@$ y $\exists k \in \{1 \dots n\},\ \alpha_k = A,\ \forall j \in \{1 \dots k-1\},\ \alpha_j = B,\ \forall j \in \{k+1 \dots n\},\ \alpha_j = I$. Tendremos que demostrar que ambos *heaps* alcanzan configuraciones equivalentes. Para simplificar y hacer más comprensible la demostración supondremos que $n = 2$ y que el primer puntero es el que se encuentra activo. La demostración con cualquier otro valor de n seguiría la misma técnica pero aplicando varias veces el paso $P0$, hasta dejar bloqueados los punteros iniciales.

Demostración de la primera implicación: $\Rightarrow$

Primero veremos en qué configuración nos quedamos cuando consumimos los n primeros punteros (en este caso $n = 2$). Realizaremos la demostración por casos sobre las posibles marcas de observación que pueda tener la ligadura $q_1^@$ equivalente.

Hipótesis:

H0 Sabemos $H_0 \equiv_R H^@$ entonces:

1. $H^@ = H'^@ + \{q_1 \overset{A}{\mapsto} q_2, q_2 \overset{I}{\mapsto} e^@\}$, o
2. $H^@ = H'^@ + \{q_1 \overset{A}{\mapsto} q_2^{@(r,s)}, q_2 \overset{I}{\mapsto} e^@\}$, o
3. $H^@ = H'^@ + \{q_1 \overset{A}{\mapsto} q_2^{@str}, q_2 \overset{I}{\mapsto} e^@\}$

Demostración:

P0.*1* Por las reglas **(demanda)** y **(activación)** ya que la regla **(demanda)** desactiva una clausura:
$H'^@ + \{q_1 \overset{A}{\mapsto} q_2, q_2 \overset{I}{\mapsto} e^@\} \overset{(demanda)}{\longrightarrow} H'^@ + \{q_1 \overset{B}{\mapsto} q_2, q_2 \overset{R}{\mapsto} e^@\} \overset{(activacion)}{\longrightarrow} H'^@ + \{q_1 \overset{B}{\mapsto} q_2, q_2 \overset{A}{\mapsto} e^@\} = H_0^@$

P0.*2* Por las reglas **(demanda@)** y **(activación)** ya que la regla **(demanda@)** desactiva una clausura:
$H'^@ + \{q_1 \overset{A}{\mapsto} q_2^{@(r,s)}, q_2 \overset{I}{\mapsto} e^@\} \overset{(demanda)}{\longrightarrow} H'^@ + \{q_1 \overset{B}{\mapsto} q_2^{@(r,s)}, q_2 \overset{R}{\mapsto} e^@\} \overset{(activacion)}{\longrightarrow} H'^@ + \{q_1 \overset{B}{\mapsto} q_2^{@(r,s)}, q_2 \overset{A}{\mapsto} e^@\} = H_0^@$

P0.*3* La regla **(observ)** nos sitúa en el apartado *2* de $P0$:
$H'^@ + \{q_1 \overset{A}{\mapsto} q_2^{@str}, q_2 \overset{I}{\mapsto} e^@\} \overset{(observ)}{\longrightarrow} H'^@ + \{q_1 \overset{A}{\mapsto} q_2^{@(n,0)}, q_2 \overset{I}{\mapsto} e^@\} = H_0^@$

Por tanto, los punteros iniciales pueden eliminarse aplicando varias veces el paso P0 correspondiente. Consideraremos el *heap* $H_0^@$ como el resultante de aplicar varias veces los pasos P0, hasta dejar activa la última clausura, la equivalente. Debemos tener en cuenta que este *heap* mantiene la equivalencia con el original; $H_0 \equiv_R H_0^@$.

(demanda)

Hipótesis:

H1 $H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q$

H2 Por la regla **(demanda)**:

$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\} = K$

Demostración:

Realizaremos la demostración por casos sobre las posibles marcas de observación que pueda tener el puntero equivalente.

P1 Por P0, H0 y H1:

1. $H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@$, o
2. $H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^{@^{@(r,s)}}$, o
3. $H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^{@^{@str}}$

P2.*1* Por la regla **(demanda)**:

$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@ \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@, p^@ \overset{B}{\mapsto} q^@\} = K^@$

P3.*1* Por H0, H2, P2.*1* y Definición 8.4:

$K \equiv_{\mathsf{R}} K^@$ $\surd(P1.1)$

P2.*2* Por la regla **(demanda@)**:

$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^{@^{@(r,s)}} \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@, p^@ \overset{B}{\mapsto} q^{@^{@(r,s)}}\} = K^@$

P3.*2* Por H0, H2, P2.*2* y Definición 8.4:

$K \equiv_{\mathsf{R}} K^@$ $\surd(P1.2)$

P3.*3* Por la regla **(observ)**:

$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^{@^{@str}} \longrightarrow H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^{@^{@(n,0)}}\}$ que nos sitúa en el apartado 2 de $P1$ $\surd(P1.3)$

(valor)

Hipótesis:

H1 $H_0 = H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q$

H2 Por la regla **(valor)**:

$H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q \longrightarrow H + \{q \overset{I}{\mapsto} w, p \overset{A}{\mapsto} w\} = K$

Demostración:

Realizaremos la demostración por casos sobre las posibles marcas de observación que pueda tener el puntero equivalente.

P1 Por P0, H0 y H1:

1. $H_0^@ = H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^@$, o
2. $H_0^@ = H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^{@^{@(r,s)}}$, o
3. $H_0^@ = H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^{@^{@str}}$

P2.*1* Por la regla **(valor)**:

$H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^@ \longrightarrow H^@ + \{q^@ \overset{I}{\mapsto} w^@, p^@ \overset{A}{\mapsto} w^@\} = K^@$

P3.*1* Por H0, H2, P2.*1* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.1)$

P2.*2* Por casos sobre w:

1. $w = \lambda x.e^{@}$, o
2. $w = \lambda^{@obs} x.e^{@}$, o
3. $w = C\ \overline{p_i^{@}}$, o

P3.*2.1* Por la regla **(valor@L)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda x.e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@^{@(r,s)}} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda x.e^{@}, p^{@} \overset{A}{\mapsto} \lambda x^{@(r,s)}.e^{@}\} = K^{@}$

P4.*2.1* Por H0, H2, P3.*2.1* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.2, P2.1)$

P3.*2.2* Por la regla **(valor@LO)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda^{@obs} x.e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@^{@(r,s)}} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda^{@obs} x.e^{@}, p^{@} \overset{A}{\mapsto} \lambda^{@(r,s):obs} x.e^{@}\} = K^{@}$

P4.*2.2* Por H0, H2, P3.2.2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.2, P2.2)$

P3.*2.3* Por la regla **(valor@C)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} C\ \overline{p_i^{@}}\} : p^{@} \overset{A}{\mapsto} q^{@^{@(r,s)}} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} C\ \overline{p_i^{@}}, \overline{q_i^{@} \overset{I}{\mapsto} p_i^{@^{@(n,i)}}}, p^{@} \overset{A}{\mapsto} C\ \overline{q_i^{@}}\} = K^{@}$

P4.*2.3* Por H0, H2, P3.2.3 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.2, P2.3)$

P3.*3* Por la regla **(observ)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} w^{@}\} : p^{@} \overset{A}{\mapsto} q^{@^{@str}} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} w^{@}\} : p^{@} \overset{A}{\mapsto} q^{@^{@(n,0)}}\}$
que nos sitúa en el apartado 2 de $P1$ $\surd(P1.3)$

(agujero negro)

Hipótesis:

H1 $H_0 = H : p \overset{A}{\mapsto} p$

H2 Por la regla **(agujero negro)**:
$H : p \overset{A}{\mapsto} p \longrightarrow H + \{p \overset{B}{\mapsto} p = K$

Demostración:

Realizaremos la demostración por casos sobre las posibles marcas de observación que pueda tener el puntero equivalente.

P1 Por P0, H0 y H1:

1. $H_0^{@} = H^{@} : p^{@} \overset{A}{\mapsto} p^{@}$, o
2. $H_0^{@} = H^{@} : p^{@} \overset{A}{\mapsto} p^{@^{@(r,s)}}$, o
3. $H_0^{@} = H^{@} : p^{@} \overset{A}{\mapsto} p^{@^{@str}}$

P2.*1* Por la regla **(agujero negro)**:
$H^{@} : p^{@} \overset{A}{\mapsto} p^{@} \longrightarrow H^{@} + \{p^{@} \overset{B}{\mapsto} p^{@}\} = K^{@}$

P3.*1* Por H0, H2, P2.*1* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.1)$

P2.*2* Por la regla **(agujero negro@)**:
$H^{@} : p^{@} \overset{A}{\mapsto} p^{@^{@(r,s)}} \longrightarrow H^{@} + \{p^{@} \overset{B}{\mapsto} p^{@^{@(r,s)}}\} = K^{@}$

P3.*2* Por H0, H2, P2.*2* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.2)$

P2.*3* Por la regla **(observ)**:
$H^{@} : p^{@} \overset{A}{\mapsto} p^{@^{@str}} \longrightarrow H^{@} + \{p^{@} \overset{A}{\mapsto} p^{@^{@(n,0)}}\}$
que nos sitúa en el apartado 2 de $P1$ $\surd(P1.3)$

(app-demanda)

Hipótesis:

H1 $H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ l$

H2 Por la regla **(app-demanda)**:
$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ l \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\ l\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^{@} = H^{@} + \{q^{@} \overset{IABR}{\mapsto} e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@}\ l^{@}$

P2 Por la regla **(app-demanda)**:
$H^{@} + \{q^{@} \overset{IABR}{\mapsto} e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@}\ l^{@} \longrightarrow H^{@} + \{q^{@} \overset{RABR}{\mapsto} e^{@}, p^{@} \overset{B}{\mapsto} q^{@}\ l^{@}\} = K^{@}$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

(**β-reducción**)

Hipótesis:

H1 $H_0 = H + \{q \overset{I}{\mapsto} \lambda x.e\} : p \overset{A}{\mapsto} q\ l$

H2 Por la regla **(β-reducción)**:
$H + \{q \overset{I}{\mapsto} \lambda x.e\} : p \overset{A}{\mapsto} q\ l \longrightarrow H + \{q \overset{I}{\mapsto} \lambda x.e, p \overset{A}{\mapsto} e[l/x]\} = K$

Demostración:

Realizaremos la demostración por casos sobre la λ-abstracción:

P1 Por P0, H0 y H1:

1. $H_0^{@} = H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda x.e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@}\ l^{@}$, o
2. $H_0^{@} = H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda^{@obs} x.e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@}\ l^{@}$

P2.*1* Por la regla **(β-reducción)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda x.e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@}\ l^{@} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} \lambda x.e^{@}, p^{@} \overset{A}{\mapsto} e^{@}[l^{@}/x]\} = K^{@}$

P3.*1* Por H0, H2, P2.*1* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd(P1.1)$

P2.*2* Por la regla (**β-reducción@**):
$H^@ + \{q^@ \overset{I}{\mapsto} \lambda x.e^@\} : p^@ \overset{A}{\mapsto} q^@\ l^@ \longrightarrow H^@ + \{q^@ \overset{I}{\mapsto} \lambda x.e^@, t \overset{I}{\mapsto} e^@[l'/x], l' \overset{I}{\mapsto} l^{@^{@(n,0)}}, p^@ \overset{A}{\mapsto} t^{@(n,1)}\} = K^@$

P3.*2* Por H0, H2, P2.*2* y Definición 8.4:
$K \equiv_{\mathsf{R}} K^@$ $\surd(P1.2)$

(letrec)

Hipótesis:

H1 $H_0 = H : p \overset{A}{\mapsto}$ **letrec** $\overline{x_i = e_i}$ **in** e

H2 Por la regla (**letrec**):

a) $H : p \overset{A}{\mapsto}$ **letrec** $\overline{x_i = e_i}$ **in** $e \longrightarrow H + \{\overline{q_i \overset{I}{\mapsto} e_i[\overline{q_i/x_i}]}, q \overset{A}{\mapsto} e[\overline{q_i/x_i}]\} = K$

b) $\overline{q_i}$ son frescos

Demostración:

P1 Por P0, H0 y H1:

a) $H_0^@ = H^@ : p' \overset{A}{\mapsto}$ **letrec** $\overline{x_i = e_i^@}$ **in** $e^@$

b) $e_i \equiv_{\mathsf{R}} e_i^@$

c) $\exists\ \eta$ en las condiciones de la definición

P2 Por la regla *Letrec*:

a) $H^@ : p' \overset{A}{\mapsto}$ **letrec** $\overline{x_i = e_i^@}$ **in** $e^@ \longrightarrow H^@ + \{\overline{q_i^@ \overset{I}{\mapsto} e_i^@[\overline{q_i^@/x_i}]}, q^@ \overset{A}{\mapsto} e[\overline{q_i^@/x_i}]\} = K^@$

b) $\overline{q_i^@}$ son frescos

P3 Por H2(b), P2(b) y P1(c):
$\eta'(x) = \begin{cases} \eta(x) & \text{si } x \neq q_i \\ q_i^@ & \text{si } x = q_i \end{cases}$ cumple las condiciones de la definición

P4 Por H0, H2, P2, P3 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^@$ $\surd$

(case-demanda)

Hipótesis:

H1 $H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto}$ **case** q **of** *alts*

H2 Por la regla (**case-demanda**):
$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto}$ **case** q **of** *alts* $\longrightarrow H + \{q \overset{RABR}{\mapsto} e,\ p \overset{B}{\mapsto}$ **case** q **of** *alts*$\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto}$ **case** $q^@$ **of** $alts^@$

P2 Por la regla (**case-demanda**):
$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto}$ **case** $q^@$ **of** $alts^@ \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@,\ p^@ \overset{B}{\mapsto}$ **case** $q^@$ **of** $alts^@\} = K^@$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

(case-reducción)

Hipótesis:

H1 $H_0 = H + \{q \overset{I}{\mapsto} C_k\ \overline{p_i}\} : p \overset{A}{\mapsto}$ **case** q **of** $\overline{C_i\ \overline{x_{ij}} \mapsto e_i}$

H2 Por la regla **(case-reducción)**:
$H + \{q \overset{I}{\mapsto} C_k\ \overline{p_i}\} : p \overset{A}{\mapsto}$ **case** q **of** $\overline{C_i\ \overline{x_{ij}} \mapsto e_i} \longrightarrow H + \{q \overset{I}{\mapsto} C_k\ \overline{p_i},\ p \overset{A}{\mapsto} e_k[\overline{p_i/x_{kj}}]\} = K$

Demostración:

P1 Por P0, H0 y H1:

a) $H_0^{@} = H^{@} + \{q^{@} \overset{I}{\mapsto} C_k\ \overline{p_i^{@}}\} : p^{@} \overset{A}{\mapsto}$ **case** $q^{@}$ **of** $\overline{C_i\ \overline{x_{ij}} \mapsto e_i^{@}}$

b) $e_i \equiv_{\mathsf{R}} e_i^{@}$

P2 Por la regla **(case-reducción)**:
$H^{@} + \{q^{@} \overset{I}{\mapsto} C_k\ \overline{p_i^{@}}\} : p^{@} \overset{A}{\mapsto}$ **case** $q^{@}$ **of** $\overline{C_i\ \overline{x_{ij}} \mapsto e_i^{@}} \longrightarrow H^{@} + \{q^{@} \overset{I}{\mapsto} C_k\ \overline{p_i^{@}},\ p^{@} \overset{A}{\mapsto} e_k[\overline{p_i^{@}/x_{kj}}]\} = K^{@}$

P3 Por H0, H2, P1(b), P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

(opP-demanda)

Hipótesis:

H1 $H_0 = H + \{\overline{q_i \overset{IABR}{\mapsto} e_i}\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n$

H2 Por la regla **(opP-demanda)**:
$H + \{\overline{q_i \overset{IABR}{\mapsto} e_i}\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n \longrightarrow H + \{\overline{q_i \overset{RABR}{\mapsto} e_i}, p \overset{B}{\mapsto} op\ \overline{q_i}^n\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^{@} = H^{@} + \{\overline{q_i^{@} \overset{IABR}{\mapsto} e_i^{@}}\} : p^{@} \overset{A}{\mapsto} op\ \overline{q_i^{@}}^n$

P2 Por la regla **(opP-demanda)**:
$H^{@} + \{\overline{q_i^{@} \overset{IABR}{\mapsto} e_i^{@}}\} : p^{@} \overset{A}{\mapsto} op\ \overline{q_i^{@}}^n \longrightarrow H^{@} + \{\overline{q_i^{@} \overset{RABR}{\mapsto} e_i^{@}}\} : p^{@} \overset{A}{\mapsto} op\ \overline{q_i^{@}}^n = K^{@}$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

(opP-reducción)

Hipótesis:

H1 $H_0 = H + \{\overline{q_i \overset{I}{\mapsto} m_i}\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n$

H2 Por la regla **(opP-reducción)**:
$H + \{\overline{q_i \overset{I}{\mapsto} m_i}\} : p \overset{A}{\mapsto} op\ \overline{q_i}^n \longrightarrow H + \{\overline{q_i \overset{I}{\mapsto} m_i}, p \overset{B}{\mapsto} op\ \overline{m_i}^n\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^@ = H^@ + \{\overline{q_i^@ \overset{I}{\mapsto} m_i}\} : p^@ \overset{A}{\mapsto} op\ \overline{q_i^@}^n$

P2 Por la regla **(opP-reducción)**:
$H^@ + \{\overline{q_i^@ \overset{I}{\mapsto} m_i}\} : p^@ \overset{A}{\mapsto} op\ \overline{q_i^@}^n \longrightarrow H^@ + \{\overline{q_i^@ \overset{I}{\mapsto} e_i^@}\} : p^@ \overset{A}{\mapsto} op\ \overline{m_i}^n = K^@$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^@\ \surd$

(seq)

Hipótesis:

H1 $H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ \texttt{`seq`}\ q'$

H2 Por la regla **(seq)**:
$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q\ \texttt{`seq`}\ q' \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\ \texttt{`seq`}\ q'\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@\ \texttt{`seq`}\ q'^@$

P2 Por la regla **(seq)**:
$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@\ \texttt{`seq`}\ q'^@ \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@, p^@ \overset{B}{\mapsto} q^@\ \texttt{`seq`}\ q'^@\} = K^@$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^@\ \surd$

(rm-seq)

Hipótesis:

H1 $H_0 = H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q\ \texttt{`seq`}\ q'$

H2 Por la regla **(rm-seq)**:
$H + \{q \overset{I}{\mapsto} w\} : p \overset{A}{\mapsto} q\ \texttt{`seq`}\ q' \longrightarrow H + \{q \overset{I}{\mapsto} w, p \overset{A}{\mapsto} q'\}$

Demostración:

P1 Por P0, H0 y H1:
$H_0^@ = H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^@\ \texttt{`seq`}\ q'^@$

P2 Por la regla **(rm-seq)**:
$H^@ + \{q^@ \overset{I}{\mapsto} w^@\} : p^@ \overset{A}{\mapsto} q^@\ \texttt{`seq`}\ q'^@ \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@, p^@ \overset{A}{\mapsto} q'^@\} = K^@$

P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^@\ \surd$

(par)

Hipótesis:

H1 $H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q \text{ ‘par‘ } q'$

H2 Por la regla **(par)**:

$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q \text{ ‘par‘ } q' \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{A}{\mapsto} q'\}$

Demostración:

P1 Por P0, H0 y H1:

$H_0^@ = H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@ \text{ ‘par‘ } q'^@$

P2 Por la regla **(par)**:

$H^@ + \{q^@ \overset{IABR}{\mapsto} e^@\} : p^@ \overset{A}{\mapsto} q^@ \text{ ‘par‘ } q'^@ \longrightarrow H^@ + \{q^@ \overset{RABR}{\mapsto} e^@, p^@ \overset{A}{\mapsto} q'^@\} = K^@$

P3 Por H0, H2, P2 y Definición 8.4:

$K \equiv_{\mathsf{R}} K^@ \surd$

Una vez demostrado que las hebras activas evolucionan a través de configuraciones equivalentes, es decir, la regla **(paralela)** mantiene la equivalencia, sólo queda por demostrar que las reglas **(desactivación)** y **(activación)** mantienen la equivalencia.

(desactivación)

Hipótesis: En este caso la clausura inicial se encuentra bloqueada, para simplificar la presentación supondremos que sólo existe una única clausura bloqueada en p:

H1 $H_0 = H + \{p \overset{AR}{\mapsto} w, q \overset{B}{\mapsto} e\}$

H2 Por la regla **(desactivación)**:

$H + \{p \overset{AR}{\mapsto} w, q \overset{B}{\mapsto} e\} \longrightarrow H + \{p \overset{I}{\mapsto} w, q \overset{R}{\mapsto} e\} = K$

Demostración:

P1 Como $H_0 \equiv_{\mathsf{R}} H^@$, considerando $N = 1$ y, por simplificación, que los punteros intermedios no poseen anotaciones de observación, entonces:

1. $H^@ = H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{A}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^@\}$ o
2. $H^@ = H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{B}{\mapsto} p_1, p_1 \overset{A}{\mapsto} w^@\}$ o
3. $H^@ = H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{B}{\mapsto} p_1, p_1 \overset{R}{\mapsto} w^@\}$ o
4. $H^@ = H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{R}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^@\}$

donde $e^@$ está bloqueada en p_0.

P2.*1* Por la regla **(valor)**:

$H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{A}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^@\}$
$\longrightarrow H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{A}{\mapsto} w^@, p_1 \overset{I}{\mapsto} w^@\}$

P3.*1* Por la regla **(desactivación)**:

$H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^@\} + \{p_0 \overset{A}{\mapsto} w^@, p_1 \overset{I}{\mapsto} w^@\}$
$\longrightarrow H'^@ + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{R}{\mapsto} e^@\} + \{p_0 \overset{I}{\mapsto} w^@, p_1 \overset{I}{\mapsto} w^@\} = K^@$

P4.*1* Por $H_0 \equiv_{\mathsf{R}} H^{@}$, H2, P3.*1* y Definición: 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$(P1.*1*)

P2.*2* Por la regla **(desactivación)**:
$H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{B}{\mapsto} p_1, p_1 \overset{A}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{R}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$

P3.*2* Por la regla **(activación)**:
$H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{R}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{A}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$

P4.*2* Por la regla **(valor)**:
$H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{A}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$
$\overset{valor}{\longrightarrow} H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{A}{\mapsto} w^{@}, p_1 \overset{I}{\mapsto} w^{@}\}$
que nos sitúa en el caso P1.*1*. $\surd$(P1.*2*)

P2.*3* Por la regla **(activación)**:
$H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{B}{\mapsto} p_1, p_1 \overset{R}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{B}{\mapsto} p_1, p_1 \overset{A}{\mapsto} w^{@}\}$
que nos sitúa en el caso P1.*2*. $\surd$(P1.*3*)

P2.*4* Por la regla **(activación)**:
$H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{R}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{q_0 \overset{B}{\mapsto} q_1, q_1 \overset{B}{\mapsto} e^{@}\} + \{p_0 \overset{A}{\mapsto} p_1, p_1 \overset{I}{\mapsto} w^{@}\}$
que nos sitúa en el caso P1.*1*. $\surd$(P1.*4*)

(activación)

Hipótesis: En este caso la clausura inicial se encuentra en estado ejecutable:

H1 $H_0 = H + \{p \overset{R}{\mapsto} e\}$

H2 Por la regla **(activación)**:
$H + \{p \overset{R}{\mapsto} e\} \longrightarrow H + \{p \overset{A}{\mapsto} e\} = K$

Demostración:

P1 Como $H_0 \equiv_{\mathsf{R}} H^{@}$, considerando $n = 2$ y, por simplificación, que los punteros intermedios no poseen anotaciones de observación, entonces:

1. $H^{@} = H'^{@} + \{p_1 \overset{R}{\mapsto} p_2, p_2 \overset{I}{\mapsto} w^{@}\}$ o
2. $H^{@} = H'^{@} + \{p_1 \overset{B}{\mapsto} p_2, p_2 \overset{R}{\mapsto} w^{@}\}$

donde $e^{@}$ está bloqueada en $p_1 = \eta p$.

P2.*1* Por la regla **(activación)**:
$H'^{@} + \{p_1 \overset{R}{\mapsto} p_2, p_2 \overset{I}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{p_1 \overset{A}{\mapsto} p_2, p_2 \overset{I}{\mapsto} w^{@}\} = K^{@}$

P3.*1* Por $H_0 \equiv_{\mathsf{R}} H^{@}$, H2, P2.1 y Definición: 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$(P1.*1*)

P2.*2* Por la regla **(activación)**:
$H'^{@} + \{p_1 \overset{B}{\mapsto} p_2, p_2 \overset{R}{\mapsto} w^{@}\}$
$\longrightarrow H'^{@} + \{p_1 \overset{B}{\mapsto} p_2, p_2 \overset{A}{\mapsto} w^{@}\} = K^{@}$
P3.*2* Por $H_0 \equiv_{\mathsf{R}} H^{@}$, H2, P2.*2* y Definición: 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$(P1.*2*)

Demostración de la segunda implicación: $\Leftarrow$

Recordaremos que $H_0 \equiv_{\mathsf{R}} H_0^{@}$. Por tanto, $H_0 = H' + \{p \overset{A}{\mapsto} e\}$ y $\exists n$, $H_0^{@} = H'^{@} + \{\overline{q_i \overset{\alpha_i}{\mapsto} q_{i+1}}^{n-1}, q_n \overset{\alpha_n}{\mapsto} e^{@}\}$ donde $rp\ e \equiv_{\mathsf{R}} rp\ e^{@}$ y $\exists k \in \{1 \ldots n\}$, $\alpha_k = A$, $\forall j \in \{1 \ldots k-1\}$, $\alpha_j = B$, $\forall j \in \{k+1 \ldots n\}$, $\alpha_j = I$. En este caso la demostración de que las hebras activas evolucionan a través de configuraciones equivalentes es más simple que la otra implicación y sigue un patron más simple. Su demostración se realiza por inducción sobre n. El caso básico $n = 1$ y el caso recursivo $n \geq 1$.

$n = 1$ En este caso al realizar un paso de cómputo en la semántica con observaciones y un paso en la semántica sin observaciones de la regla equivalente (es decir, si hemos aplicado la regla **(agujero negro@)** o **(agujero negro)** la equivalente sin marcas de observación es la regla **(agujero negro)**), alcanzamos directamente una configuración equivalente. La única excepción consiste en la regla **(observ)**, que genera configuraciones equivalentes sin necesidad de dar ningún paso en la semántica de GpH sin observaciones.

Como caso particular lo demostraremos para la regla **(demanda@)**. Las demás siguen el mismo patrón simple y se demuestran de la misma forma, por eso no se presentan aquí.

(demanda@)

Hipótesis:

H0 $H_0 \equiv_{\mathsf{R}} H_0^{@}$
H1 $H_0^{@} = H^{@} + \{q^{@} \overset{IABR}{\mapsto} e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@(r,s)}$
H2 Por la regla **(demanda@)**:
$H^{@} + \{q^{@} \overset{IABR}{\mapsto} e^{@}\} : p^{@} \overset{A}{\mapsto} q^{@(r,s)} \longrightarrow H^{@} + \{q^{@} \overset{RABR}{\mapsto} e^{@}, p^{@} \overset{B}{\mapsto} q^{@(r,s)}\} = K^{@}$

Demostración:

P1 Por H0 y H1:
$H_0 = H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q$
P2 Por la regla **(demanda)**:
$H + \{q \overset{IABR}{\mapsto} e\} : p \overset{A}{\mapsto} q \longrightarrow H + \{q \overset{RABR}{\mapsto} e, p \overset{B}{\mapsto} q\} = K$
P3 Por H0, H2, P2 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

$n \geq 1$ Lo único necesario es dar un paso en la semántica con observaciones, sin darlo en la configuración equivalente sin observaciones. Este paso inicial corresponderá a la

aplicación de alguna de las siguientes reglas: **(observ)**, **(demanda)** junto con la regla **(activación)**, **(demanda@)** junto con la regla **(activación)**, **(valor)**, **(valor@C)**, **(valor@L)** y **(valor@LO)**.

Como caso particular lo demostraremos para la regla **(demanda@)** junto con la regla **(activación)**. Las demás siguen el mismo patrón simple y se demuestran de la misma forma, por eso no se presentan aquí.

(demanda@)

Hipótesis:

H0 $H_0 \equiv_{\mathsf{R}} H_0^{@}$

H1 $H_0^{@} = H'^{@} + \{\overbrace{q_i \overset{\alpha_i}{\mapsto} q_{i+1}}^{n-1},\ q_n \overset{\alpha_n}{\mapsto} e^{@}\}$ donde $rp\ e \equiv_{\mathsf{R}} rp\ e^{@}$ y $\exists k \in \{1 \ldots n\}$ tal que:

a) $\alpha_k = A$

b) $\forall j \in \{1 \ldots k-1\},\ \alpha_j = B$

c) $\forall j \in \{k+1 \ldots n\},\ \alpha_j = I$

Demostración:

P1 Por la regla **(demanda@)** sobre q_k:
$H^{@} + \{q_{k+1} \overset{I}{\mapsto} q_{k+2}\} : q_k \overset{A}{\mapsto} q_{k+1}^{@(r,s)} \longrightarrow H^{@} + \{q_{k+1} \overset{R}{\mapsto} q_{k+2}, q_k \overset{B}{\mapsto} q_{k+1}^{@(r,s)}\}$

P2 Por la regla **(activación)** ya que la regla **(demanda@)** desactiva una clausura:
$H^{@} + \{q_{k+1} \overset{R}{\mapsto} q_{k+2}, q_k \overset{B}{\mapsto} q_{k+1}^{@(r,s)}\} \longrightarrow H^{@} + \{q_{k+1} \overset{A}{\mapsto} q_{k+2}, q_k \overset{B}{\mapsto} q_{k+1}^{@(r,s)}\} = K^{@}$

P3 Por H0, H1, P2 y Definición 8.4:
$H_0 \equiv_{\mathsf{R}} K^{@}$ $\surd$

Demostración 15 (Proposición 8.2) En este caso la demostración se realizará en tres pasos:

1. Ver que las reglas de evaluación local partiendo de *heaps* equivalentes nos llevan a *heaps* equivalentes, es decir, analizar la evolución independiente de cada proceso. Este paso ya está demostrado, salvo para los *streams*, ya que cuando no existen *streams* las reglas locales de Eden son exactamente las mismas que las de GpH. La única diferencia con GpH radica en que ahora una ligadura del *heap* nunca puede encontrarse en estado ejecutable (su equivalente en Eden es activo). Por tanto, tanto la regla de **(activación)** como la regla de **(desactivación)** en Eden son más simples y corresponden con las reglas de **(desbloqueo de *whnf*)** y **(desactivación de *whnf*)** respectivamente.

   Veremos que los *streams* mantienen la equivalencia. Para ello demostraremos que las reglas **(demanda-*stream*)** y **(demanda de la comunicación de la cabeza del *stream*)** mantienen la equivalencia. Veremos sólo la implicación hacia la derecha ya que la otra implicación es bastante más simple. Partiremos de que ya hemos consumido los punteros iniciales, tal y como se hizo en la demostración de GpH.

2. Ver que la función nh genera *heaps* equivalentes cuando se parten de *heaps* equivalentes. Con esto concluiremos que las reglas de transmisión de datos, es decir, **(comunicación de valores)**, **(comunicación *stream*-vacío)** y **(comunicación de la cabeza del *stream*)** mantienen la relación de equivalencia.

3. Ver que la regla **(creación de proceso)** y la regla **(creación de proceso@)** son equivalentes bajo la relación de equivalencia $\equiv_{\mathsf{R}}$,

Pasemos, por tanto, a demostrar cada uno de los puntos:

1. Demostraremos que las reglas de los *streams* evolucionan a través de configuraciones equivalentes. Sólo demostraremos la implicación $\Rightarrow$ habiendo consumido los punteros iniciales:

   (demanda-*stream*)

   Hipótesis:

   H0 $H_0 \equiv_{\mathsf{R}} H^{@}$ y $H_0^{@}$ corresponde con la evolución de los punteros iniciales de $H^{@}$

   H1 $H_0 = H + \{p_1 \overset{IAB}{\mapsto} e\} : ch \overset{A}{\mapsto} [p_1 : p_2]$

   H2 Por la regla **(demanda-*stream*)**:
   $H + \{p_1 \overset{IAB}{\mapsto} e\} : ch \overset{A}{\mapsto} [p_1 : p_2] \longrightarrow H + \{p_1 \overset{AAB}{\mapsto} e, ch \overset{B}{\mapsto} [p_1 : p_2]\} = K$

   Demostración:

   P1 Por H0 y H1:
   $H_0^{@} = H^{@} + \{p_1^{@} \overset{IAB}{\mapsto} e^{@}\} : ch^{@} \overset{A}{\mapsto} [p_1^{@} : p_2^{@}]$

   P2 Por la regla **(demanda-*stream*)**:
   $H^{@} + \{p_1^{@} \overset{IAB}{\mapsto} e^{@}\} : ch^{@} \overset{A}{\mapsto} [p_1^{@} : p_2^{@}] \longrightarrow H^{@} + \{p_1^{@} \overset{AAB}{\mapsto} e^{@}, ch^{@} \overset{B}{\mapsto} [p_1^{@} : p_2^{@}]\} = K^{@}$

   P3 Por H0, H2, P2 y Definición 8.4:
   $K \equiv_{\mathsf{R}} K^{@}$ $\surd$

   (demanda de la comunicación de la cabeza del *stream*)

   Hipótesis:

   H0 $H_0 \equiv_{\mathsf{R}} H_0^{@}$

   H1 $H_0 = H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2]\}$

   H2 Por la regla **(demanda de la comunicación de la cabeza del *stream*)**
   $(p \overset{I}{\mapsto} e \in \mathsf{nff}(w, H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2]\}))$:
   $H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2]\} \longrightarrow H + \{p_1 \overset{I}{\mapsto} w, ch \overset{IB}{\mapsto} [p_1 : p_2], p \overset{A}{\mapsto} e\} = K$

   Demostración:

   P1 Por H0 y H1:
   $H_0^{@} = H^{@} + \{p_1^{@} \overset{I}{\mapsto} w^{@}, \overline{q_i \overset{IB}{\mapsto} q_{i+1}}^{N-1}, q_N \overset{IB}{\mapsto} [p_1^{@} : p_2^{@}]\}$

   P2 Por la regla **(demanda de la comunicación de la cabeza del *stream*)**
   $(p^{@} \overset{I}{\mapsto} e^{@} \in \mathsf{nff}(w^{@}, H_0^{@}))$:
   $H^{@} + \{p_1^{@} \overset{I}{\mapsto} w^{@}, \overline{q_i \overset{IB}{\mapsto} q_{i+1}}^{N-1}, q_N \overset{IB}{\mapsto} [p_1^{@} : p_2^{@}]\}\} \longrightarrow H^{@} + \{p_1^{@} \overset{I}{\mapsto} w^{@}, \overline{q_i \overset{IB}{\mapsto} q_{i+1}}^{N-1}, q_N \overset{IB}{\mapsto} [p_1^{@} : p_2^{@}], p^{@} \overset{A}{\mapsto} e^{@}\} = K^{@}$

   P3 Por definición de nff:
   Los punteros p y $p^{@}$ cumplen la Definición 8.4, es decir, son los equivalentes.

P4 Por H0, H2, P2, P3 y Definición 8.4:
$K \equiv_{\mathsf{R}} K^{@}$ $\surd$

Por tanto, queda demostrado el primer punto de la equivalencia en Eden.

2. Demostraremos que la función nh genera *heaps* equivalentes:

nh genera *heaps* equivalentes

Hipótesis:

H1 Sean $H \equiv_{\mathsf{R}} H^{@}$

H2 Sea e una expresión y η el renombramiento de H1 tal que $e^{@} = \eta\ e$, veremos que $\mathsf{rch}(H, e) \equiv_{\mathsf{R}} \mathsf{rch}(H^{@}, e^{@})$

Demostración:

P1 Por H1:
$\eta\ \mathsf{rch}(H, e) \subseteq \mathsf{rch}(H^{@}, e^{@})$

P2 Por definición de la función rch los *heaps* generados son correctos, es decir, contienen todas las variables libres de las expresiones almacenadas en ellos.

P3 La función mo no influye sobre la equivalencia.

P4 Por P1, P2, P3 y debido a que la función nh deja las clausuras inactivas:
$\mathsf{nh}(id, e, H) \equiv_{\mathsf{R}} \mathsf{nh}(id, e^{@}, H^{@})$

Como nh genera *heaps* equivalentes y las reglas **(comunicación de valores)**, **(comunicación *stream*-vacío)** y **(comunicación de la cabeza del *stream*)** sólo comunican el *heap* necesario, es decir, $\mathsf{nh}(id_1, w, H_1)$ concluimos que dichas reglas mantienen la relación de equivalencia.

3. Demostremos ahora que las reglas **(creación de proceso)** y **(creación de proceso@)** generan configuraciones equivalentes:

(creación de proceso) y (creación de proceso@) mantienen la equivalencia

Hipótesis:

H1 Sea $H + \{p \overset{\alpha}{\mapsto} q\#l\} \equiv_{\mathsf{R}} H^{@}$

H2 Por la regla **(creación de proceso)**:

a) $H\ q = \lambda x.e$

b) $\Big(S, \langle id, H + \{p \overset{\alpha}{\mapsto} q\#l\}\rangle\Big) \overset{pcu}{\longrightarrow}$
$\left(S,\ \begin{array}{l}\langle id, H + \{p \overset{B}{\mapsto} ch_o, ch_i \overset{A}{\mapsto} l\} = K\rangle, \\ \langle id', \eta'(\mathsf{nh}(id, q, H)) + \{ch_o \overset{A}{\mapsto} \eta'(q)\, l', l' \overset{B}{\mapsto} ch_i\}\rangle\end{array}\right)$
donde $frescas(id', ch_i, ch_o, l')$

Demostración: Realizaremos la demostración sólo cuando $H^{@}$ evoluciona aplicando la regla **(creación de proceso@)**. La demostración cuando $H^{@}$ evoluciona a través de la regla **(creación de proceso)** es más simple y muy similar a la que aquí se presenta.

P1 Por H1:
$\exists N, H^{@} = H'^{@} + \{\overline{p_i \overset{\alpha_i}{\mapsto} p_{i+1}}^{N-1}, p_N \overset{\alpha_N}{\mapsto} e^{@}\}$ donde $rp\ e \equiv_{\mathsf{R}}\ rp\ e^{@}$

P2 Por P1:
$e^{@} = q^{@}\#l^{@}$

P3 Por P2:
$H^{@}\ q^{@} = \lambda^{@[\overline{(r_i,s_i)}]}x.e^{@}$

P5 Por la regla **(creación de proceso@)**:
$\left(S^{@}, \langle id^{@}, H^{@} + \{p_N \overset{\alpha_N}{\mapsto} q^{@}\#l^{@}\}\rangle\right) \xrightarrow{pc@}$
$$\left(S^{@},\ \begin{array}{l}\langle id^{@}, H^{@} + \left\{\begin{array}{l} p_N \overset{B}{\mapsto} p'^{@(length\ f,1)}, p' \overset{B}{\mapsto} ch_o^{@}, \\ ch_i^{@} \overset{A}{\mapsto} l^{@@(length\ f,0)}\end{array}\right\} = K^{@} \looparrowright f \circ \langle\overline{[(r_i\ s_i)]}\ Fun\rangle\rangle, \\ \langle id'^{@}, \eta''(\mathsf{nh}(id^{@}, q^{@}, H^{@})) + \{ch_o^{@} \overset{A}{\mapsto} \eta''(q^{@})\, l'^{@}, l'^{@} \overset{B}{\mapsto} ch_i^{@}\}\rangle\end{array}\right)$$
donde $fresh(id'^{@}, ch_i^{@}, ch_o^{@}, l'^{@}, p')$

P6 Por nh genera *heaps* equivalentes:
$\mathsf{nh}(id, q, H) \equiv_{\mathsf{R}}\ \mathsf{nh}(id^{@}, q^{@}, H^{@})$

P7 Por P6, $fresh(ch_o, l')$, $fresh(ch_o^{@}, l'^{@})$ y Definición 8.4 (los punteros extras son "de paso"):
$\eta'(\mathsf{nh}(id, q, H)) + \{ch_o \overset{A}{\mapsto} \eta'(q)\, l', l' \overset{B}{\mapsto} ch_i\} \equiv_{\mathsf{R}}\ \eta''(\mathsf{nh}(id^{@}, q^{@}, H^{@})) + \{ch_o^{@} \overset{A}{\mapsto} \eta''(q^{@})\, l', l' \overset{B}{\mapsto} ch_i^{@}\}$ $\surd$(los *heaps* de id' e $id'^{@}$ mantienen la equivalencia)

P8 Por H1, $fresh(ch_o, ch_i)$, $fresh(ch_o^{@}, ch_i^{@}, p'^{@})$, y Definición 8.4 ($p'^{@}$ es "de paso"):
$K \equiv_{\mathsf{R}}\ K^{@}$ $\surd$(los *heaps* de id e $id^{@}$ mantienen la equivalencia)

Resumiendo, por un lado hemos visto que dando más pasos de cómputos en la semántica con observaciones alcanzamos configuraciones equivalentes. (En programación paralela la actividad de dos hebras antes de la sincronización mantiene la equivalencia de cómputos.) Por otro lado, hemos visto que la comunicación de valores (momento en el cual los procesos se sincronizan en Eden) mantiene la equivalencia. Como la comunicación fuerza a que los datos a enviar se encuentren en *whnf*, se transmiten los mismos datos. Finalmente, hemos visto que la creación de procesos genera *heaps* equivalentes tanto si los creamos a partir de la regla **(creación de proceso)**, como de la regla **(creación de proceso@)**. Cabe concluir que se cumple la Proposición 8.2.

Bibliografía

[ACCL90] M. Abadi, L. Cardelli, P. L. Curien, and J.-J. Lèvy. Explicit substitutions. In *7th Annual ACM Symposium on Principles of Programming Languages (POPL 1990)*, pages 31–46. ACM, 1990.

[Aug84] L. Augustsson. A compiler for Lazy ML. In *1984 ACM Symposium on Lisp and Functional Programming (LFP 1984)*, pages 218–227. ACM, 1984.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, 2nd edition, 1998.

[BKT00] C. Baker-Finch, D. J. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 162–173. ACM, 2000.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden, the paradise of functional-concurrent programming. In *2nd International European Conference on Parallel Processing (EUROPAR 1996)*, volume 1123 of *LNCS*, pages 710–713. Springer, 1996.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *2nd International Workshop on High-level Parallel Programming Models (HIPS 1997)*, pages 120–124. IEEE Computer Science Press, 1997.

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical report, Philipps-Universität, 1998.

[BP99] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[Bru72] N. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34(5):381–392, 1972.

[Bru78] N. De Bruijn. Lambda calculus notation with namefree formulas involving symbols that represent reference transforming mapping. *Indag. Math.*, 40:348–356, 1978.

[Cab05] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *2005 ACM SIGPLAN Workshop on Curry and functional logic programming (WCFLP 2005)*, pages 8–13. ACM, 2005.

[CKLP01] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal - nested data parallelism in Haskell. In *7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par 2001)*, volume 2150 of *LNCS*, pages 524–534. Springer, 2001.

[CRW01] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In *12th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *LNCS*, pages 176–193. Springer, 2001.

[CRW02] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In *14th International Workshop on the Implementation of Functional Languages (IFL 2002)*, volume 2670 of *LNCS*, pages 165–181. Springer, 2002.

[Cur91] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[ELR05] A. Encina, L. Llana, and F. Rubio. Formalizing the debugging process in Haskell. In *2nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2005)*, volume 3722 of *LNCS*, pages 211–226. Springer, 2005.

[ELR06a] A. Encina, L. Llana, and F. Rubio. Introducing debugging capabilities to natural semantics. In *6th International Andrei Ershov Memorial Conference, Perspectives of System Informatics (PSI 2006)*, volume 4378 of *LNCS*, pages 195–208. Springer, 2006.

[ELR06b] A. Encina, L. Llana, and F. Rubio. Introducing debugging capabilities to natural semantics (extended version). Available from authors, 2006.

[ELRH07] A. Encina, L. Llana, F. Rubio, and M. Hidalgo-Herrero. Observing intermediate structures in a parallel lazy functional language. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pages 109–120. ACM, 2007.

[Enc01] A. Encina. Generación de código para lenguajes funcionales perezosos. Master's thesis, Universidad Complutense de Madrid, 2001.

[EP01] A. Encina and R. Peña. Proving the correctness of the STG machine. In *13th International Workshop on the Implementation of Functional Languages (IFL 2001)*, volume 2312 of *LNCS*, pages 88–104. Springer, 2001.

[EP03a] A. Encina and R. Peña. Formally deriving an STG machine. In *5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2003)*, pages 102–112. ACM, 2003.

[EP03b] R. Ennals and S. L. Peyton Jones. HsDebug: debugging lazy programs by not being lazy. In *2003 ACM SIGPLAN Workshop on Haskell (Haskell 2003)*, pages 84–87. ACM, 2003.

[EP08] A. Encina and R. Peña. From natural semantics to C: A formal derivation of two STG machines. Accepted in *Journal of Functional Programming*, 2008.

[ERR07] A. Encina, I. Rodríguez, and F. Rubio. Testing speculative work in a lazy/eager parallel functional language. In *Languages and Compilers for Parallel Computing (LCPC 2005)*, volume 4339 of *LNCS*, pages 274–288. Springer, 2007.

[Fin99] S. Finne, editor. *HaskellDirect user's manual*. `http://www.haskell.org/hdirect/`, Noviembre 1999.

[FLMP99] S. Finne, D. Leijen, E. Meijer, and S. L. Peyton Jones. Calling hell from heaven and heaven from hell. In *4th ACM SIGPLAN International Conference on Functional Programming (ICFP 1999)*, pages 114–125. ACM, 1999.

[FW87] J. Fairbairn and S. C. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *1987 Conference on Functional Programming Languages and Computer Architecture (FPCA 1987)*, volume 274 of *LNCS*, pages 34–45. Springer, 1987.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[Gil00a] A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*, page 12 pages. Technical report, University of Nottingham, 2000.

[Gil00b] A. Gill. Hood homepage. `http://www.haskell.org/hood/`, 2000.

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. The MIT Press, 1999.

[Her00] C. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Universidad de Passau, 2000.

[Hid04] M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. PhD thesis, Universidad Complutense de Madrid, 2004.

[Hil94] J. M. D. Hill. *Data-Parallel Lazy Functional Programing*. PhD thesis, Universidad de Londres, 1994.

[Him06] D. Himmelstrup. Interactive debugging with GHCi. In *2006 ACM SIGPLAN workshop on Haskell (Haskell 2006)*, pages 107–107. ACM, 2006.

[HL00] J. Holmerin and B. Lisper. Development of parallel algorithms in data field Haskell (research note). In *6th International Euro-Par Conference on Parallel Processing (Euro-Par 2000)*, volume 1900 of *LNCS*, pages 762–766. Springer, 2000.

[HO02] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

[HOR07] M. Hidalgo-Herrero, Y. Ortega-Mallén, and F. Rubio. Comparing alternative evaluation strategies for stream-based parallel functional languages. In *18th International Symposium on Implementation and Application of Functional Languages (IFL 2006)*, volume 4449 of *LNCS*, pages 55–72. Springer, 2007.

[HPR00] F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *1st Scottish Functional Programming Workshop (SFP 200)*, pages 11–19. Intellect, 2000.

[Hug89] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Hug90] J. Hughes. Why functional programming matters. In *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[IM07] J. Iborra and S. Marlow. Examine your laziness. A lightweight procedural debugging technique for Haskell. Technical report, Universidad Politécnica de Valencia, April 2007.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984.

[Kel87] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Westfield College, 1987.

[KLPR01] U. Klusik, R. Loogen, S. Priebe, and F. Rubio. Implementation skeletons in Eden: Low-effort parallel programming. In *18th International Symposium on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *LNCS*, pages 71–88. Springer, 2001.

[KLS94] C. H. Koelbel, D. B. Loveman, and R. S. Schreiberand. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. The MIT Press, 1994.

[KPR00] U. Klusik, R. Peña, and F. Rubio. Replicated workers in Eden. In *2nd International Workshop on Constructive Methods for Parallel Programming (CMPP 2000)*, pages 143–164. Nova Science, 2000.

[Lau82] M. Lauer. Computing by homomorphic images. In *Computer algebra: symbolic and algebraic computation*, pages 139–168. Springer, 1982.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1993)*. ACM, 1993.

[Lip71] J. D. Lipson. Chinese remainder and interpolation algorithms. In *2nd ACM symposium on Symposium on Symbolic and Algebraic Manipulation (SYMSAM 1971)*, pages 372–391. Academic Press, 1971.

[LOP+02] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*, pages 95–128. Springer, 2002.

[LOP05] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[LRS+03] H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[MFH95] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *7th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1995)*, pages 66–77. ACM, 1995.

[MK06] O. Mürk and L. Kolmodin. Rectus - locally eager haskell debugger. Technical report, Göteborg University, 2006.

[MP04] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *SIGPLAN Not.*, 39(9):4–15, 2004.

[MP06] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[MSB05] I. MacLarty, Z. Somogyi, and M. Brown. Divide-and-query and subterm dependency tracking in the mercury declarative debugger. In *6th international symposium on Automated analysis-driven debugging (AADEBUG 2005)*, pages 59–68. ACM, 2005.

[NA01] R. S. Nikhil and Arvind. *Implicit Parallel Programming In PH.* Academic Press. Morgan Haufman Publishers, 2001.

[Nil01] H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[Nil07] H. Nilsson. Freja homepage. `http://www.ida.liu.se/~henni/`, 2007.

[NS97] H. Nilsson. and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 2(2):121–150, 1997.

[NW98] G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1–2):49–98, 1998.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Pey92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, version 2.5. *Journal of Functional Programming*, 2(2):127–202, 1992.

[Pey96] S. L. Peyton Jones. Compiling Haskell by program transformations: A report from the trenches. In *6th European Symposium on Programming (ESOP 1996)*, LNCS. Springer, 1996.

[PHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993.

[Pit05] A. Pitts. Alpha-structural recursion and induction. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOL 2005)*, volume 3603 of *LNCS*, pages 17–34. Springer, 2005.

[PL91] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *5th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1991)*, volume 523 of *LNCS*, pages 636–666. Springer, 1991.

[PL92] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages, A Tutorial*. Prentice-Hall, 1992.

[Pop98] B. Pope. *Buddha: A declarative debugger for Haskell*. PhD thesis, Department of Computer Science, University of Melbourne, 1998.

[Pop04] B. Pope. Declarative debugging with Buddha. In *5th International School on Advanced Functional Programming (AFP 2004)*, volume 3622 of *LNCS*, pages 273–308. Springer, 2004.

[Pop07] B. Pope. Buddha homepage. `http://www.cs.mu.oz.au/~bjpop/buddha/`, 2007.

[PR01] R. Peña and F. Rubio. Parallel functional programming at two levels of abstraction. In *3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 187–198. ACM, 2001.

[PS89] S. L. Peyton Jones and J. Salkild. The Spineless Tagless G-machine. In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989)*, pages 184–201. ACM, 1989.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. LU Decomposition and Its Applications. In *Numerical Recipes in C: The Art of Scientific Computing*, pages 43–49. Cambridge University Press, 1992.

[Rei01] C. Reinke. GHood — graphical visualization and animation of Haskell object observations. In *Proceedings of the 5th Haskell Workshop*, volume 59 of *ENTCS*. Elsevier Science, 2001.

[Rei07] C. Reinke. Ghood homepage. `http://www.cs.kent.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`, 2007.

[Rub01] F. Rubio. *Programación funcional paralela eficiente en Eden*. PhD thesis, Departamento de Sistemas Informáticos y Programación. Universidad Complutense de Madrid, 2001.

[Run07] C. Runciman. Hat homepage. `http://www.haskell.org/hat/`, 2007.

[Seg01] C. Segura. *Análisis de programas en lenguajes funcionales paralelos.* PhD thesis, Departamento de Sistemas Informáticos y Programación. Universidad Complutense de Madrid, 2001.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[Tes96] A. Tessier. Declarative debugging in constraint logic programming. In *2nd Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security (ASIAN 1996)*, LNCS, pages 64–73. Springer, 1996.

[THJ+96] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1996)*, pages 79–88. ACM, 1996.

[THLP98] P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[TLP02] P. W. Trinder, H.W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.

[UPG04] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[WCBR01] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *2001 ACM SIGPLAN Haskell Workshop*, volume 59(2) of *ENTCS*, pages 151–170. Elsevier Science, 2001.